National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/ i) c

Canada

The University of Alberta

A COST MODEL FOR COMPUTING

by

Charles van Duren

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Computing Science

Edmonton, Alberta
Spring, 1989

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-52988-1

Canada

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Charles van Duren

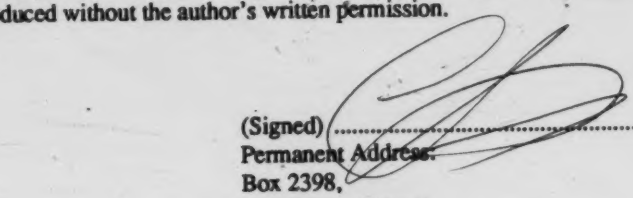
TITLE OF THESIS: A Cost Model for Computing

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1989

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.



(Signed) .....
Permanent Address:
Box 2398,
Athabasca, Alberta
T0G 0B0

Dated December 21, 1988

THE UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **A Cost Model for Computing** submitted by **Charles van Duren** in partial fulfillment of the requirements for the degree of **Master of Science.**

.....

Supervisor

.....

.....

.....

Date

ABSTRACT

Complexity analysis facilitated great progress in computing science. The abstract computational model and the sound foundation in mathematics provided the means for assessing and guiding developments in existing and new algorithms. Unfortunately, theory and practice increasingly diverged with technological developments in computing systems, and the links between algorithms and their implementations, and therefore between the predicted performance of the latter, became increasingly tenuous. The performance of programs is difficult, if not impossible, to predict. Complexity analysis is based on an abstract model of computation devoid of implementation detail. Programs specify all implementation details required for execution in real computing environments, and operate on finite quantities of 'average' data.

The diversity among, and complexity of modern computing environments militates against the development of an effective, abstract model of computing which reflects the costs of program execution in such environments. Using the systems science concepts of interface, structure, and dynamics, this thesis examines modern complex computing systems to determine the basis for a cost model of program execution.

The base model postulates memory references as the dominant operations. Examination of technological developments and the implementation of memory hierarchies provide evidence for this. Application of the model to performance improvement takes the form of minimization of memory references. Initial experimental evidence suggests that the general form of this claim is justified.

Additional evidence suggests that the model may be able to predict the relative performance of programs and evaluate proposed improvements. The approach taken by this thesis suggests that such a general model may be possible, and that the science basis of computing science be expanded to the experimental investigation of algorithms and programs so that such a model may become a reality.

Acknowledgements

I want to thank the Department of Computer Science for providing me with the facilities to do this research. In am particularly thankful to the administrative staff for smoothing out the day-to-day problems associated with student life. Academic advice from various faculty members was also appreciated.

I am especially thankful to Wlodek and Yan for providing me with a home away from my home in the "barren north" country of Athabasca. Their hospitality and friendship was much appreciated.

Thanks to ITT Business Systems, where working conditions convinced me to return to school. Also to Greyhound, for carrying me back and forth every week.

Last but not least, I thank Judith for helping to maintain my sanity during the 20 months that this work kept us from living a normal life.

Table of Contents

Chapter	Page
Chapter 1: Introduction	1
1.1. The Problem Determined	1
1.2. Outline of the Thesis	3
Chapter 2: Program Performance	5
2.1. Introduction	5
2.2. Complexity Analysis	5
2.2.1. Computational Models for Complexity Analysis	6
2.3. Puzzling Evidence	8
2.3.1. The Case of The Wandering Salesman	8
2.3.2. Two Wrongs Don't Make a Right	9
2.4. Mapping Computations to Processes	10
2.5. Conclusions	11
Chapter 3: Performance in Systems with Memory Hierarchies	14
3.1. Introduction	14
3.2. Memory Hierarchies	14
3.3. Effects on Performance	15
3.4. Program Transformation	16
3.5. Considerations in Modeling Computing Environments	18
3.6. Conclusions	19
Chapter 4: Description and Analysis of Computing Systems	20

4.1. Introduction 20
4.1.1. Analysis of Complex Systems 22
4.2. The Systems Approach 22
4.2.1. Analysis and Description of Systems 23
4.3. Computer Systems Design 25
4.4. Virtual Machines 26
4.5. Virtual Machines and Instruction Sets 28
4.6. Computers as Complex Hierarchical Systems 30
4.7. Conclusion 30
Chapter 5: A Cost Model for Computing 31
5.1. Introduction 31
5.2. The Cost Model 31
5.2.1. Interface (Environment) 31
5.2.1.1. Computers as Complex Artificial Systems 32
5.2.2. Structure 37
5.2.2.1. The von Neumann Model 37
5.2.2.2. Memory Limitations 40
5.2.3. Dynamics 42
5.2.3.1. Algorithms and Execution in Abstract Environments 42
5.2.3.2. Relative Cost of Instructions 46
5.3. Conclusion 48
Chapter 6: Application of the Model 49

6.1. Introduction 49

6.2. Effects of Memory Hierarchy on Performance 50

6.3. Cost of Memory and Processor Operations 51

6.4. The Model Applied 54

6.4.1. Case Study 1 - Integration with Other Improvements 55

6.5. Case Study 2 - Memory Reference Minimization 60

6.6. Performance Analysis and Prediction 62

6.7. Conclusions 64

Chapter 7: Evaluation and Conclusions 67

7.1. Introduction 67

7.2. Model Validity 68

7.3. Application of the Model 70

7.4. Further Research Directions 71

7.5. Conclusion 72

References 74

List of Figures

Figure	Page
2.1 Unpredictability of Behaviour: Traveling Salesman Problem	8
2.2 Mapping Algorithms to Executable Code	10
4.1 Logical system view	27
5.1 Logical Model - Controlled Manipulation of Memory	38
5.2 Von Neumann Model As Implemented	40
5.3 Hierarchical Structure of Computer Problem Solution	42
5.4 Structure of Problem Solution in Multi-programming Environment	43
5.5 Program in execution view of memory	44
6.1 Performance in (Simulated) One-level Memory	50
6.2 Performance in (Simulated) Memory Hierarchy	50
6.3 Relative Cost of Operations (VAX-11/780)	51
6.4 Relative Cost of Operations (MIPS)	53
6.5 Case Study 1: Traveling Salesman (VAX-11/780)	55
6.6 Case Study 2: Traveling Salesman (MIPS M/1000)	61
6.7 Cost Model's Performance Prediction (Case Study 1)	62
6.8 Cost Model's Performance Prediction (Including Locality)	64

Chapter 1

Introduction

1.1. The Problem Determined

Programming, from its inception, has been concerned with performance in execution. Early programs controlled the hardware "state" of the machine through the manipulation of bits. An early programming language, such as Fortran, closely resembled in its structures and operations the underlying machine architecture and language. Effects of changes in algorithms or structures used were thus reflected almost directly in the performance of programs, particularly since these were executed on a dedicated, single-user machine.

Vast improvements in the speed, size, and capacity of computers have not diminished the need for concern with performance issues. Programs have increased greatly in size and complexity, and many applications have been found for them in time-sensitive applications, and to problems of unprecedented size and difficulty. A number of factors have contributed, however, to a diminished "reality" (compared to early bit manipulation) in programs. The relationship between statements in a program and changes of state in the machine effected by their execution is no longer easily discerned, if perceptible at all.

Not the least of these developments was the advent and popularity of high-level languages, which changed the focus of programming activity from the manipulation of the machine for various ends to the statement and solution of solutions to problems. Languages designed for specific classes of applications (e.g. Fortran, Cobol) emphasized the shift away from the machine. Implementation details were left to interpreters, compilers, and operating systems. A language such as LISP makes heavy use of system functions invisible to the programmer to effect its list-processing capabilities, while environments such as Smalltalk are designed specifically to hide implementation details deemed non-essential to the problem-solving process.

A parallel development was the emergence of a whole range of software engineering techniques concerned with the production of software systems. As program development and maintenance became the dominant cost in systems development, the emphasis in software development shifted to making programs easier to write, understand, and change. The development process stresses structured techniques that tend to favour correctness and maintainability over efficiency. There is little doubt that the diminished emphasis on efficiency led to the production of systems whose performance could be greatly improved.

The development of sophisticated computing environments, with such features as memory hierarchies, multi-programming, and multi-processor capabilities contributed substantially to the alienation of the programmer from the machine as hardware. Programs are written for virtual machines several logical levels removed from the executing hardware, and the complexity of the hardware and intervening software levels makes it difficult, to say the least, to establish any connection between operations specified by a particular program and the resulting changes of state in the host machine.

The increasing sophistication and complexity of computing machines and the growing abstraction from the machine of the languages used to program them have made it increasingly difficult to make meaningful statements about computing. In particular, the prediction of the performance of high-level language programs suffers from the lack of a consistent model of computing on which to base its analysis and prediction. Since any determination of a cost function for program execution must be derived from the interface between computer and program, what is needed is a model of computation based on realistic computing environments and of processes (programs in execution) in such environments.

1.2. Outline of the Thesis

The goal of this thesis is to investigate the possibility of modeling complex computing environments for the purpose of deriving a standard for high-level language program performance analysis and improvement. The approach chosen is to postulate a base model reflecting design considerations for and empirical evidence about the functioning of such implemented environments. The perspective, or base model, is that of user-as-programmer, defining the interface to be the high-level language virtual machine level.

The first requirement of model-building is the clear, logical, unambiguous statement of the problem to be investigated. This is accomplished through an analysis of present approaches to performance analysis and/or prediction in Chapter 2. We consider complexity analysis, and find it wanting for our needs because of its deliberate exclusion of implementation considerations. In Chapter 3 we consider how implementation, particularly of memory hierarchies, is of primary concern when considering performance. Some requirements for modeling computing are discussed.

The methodology for describing and analyzing computing systems is discussed in Chapter 5. Complexity can be reduced and managed through structure imposed using knowledge about organization and behaviour. Systems science provides the conceptual framework for the analysis of the behaviour of the system. We consider this approach to reducing complexity, and in particular its application in computer design methodologies. A major element in these methodologies is the concept of the virtual machine, providing organization and a conceptual framework for the design and analysis of complex systems.

The cost model for program execution is formulated in Chapter 4. The systems analysis categories of interface, structure, and dynamics are applied to the description of the computing environment as perceived by and accessible to the user as programmer. High-level languages are the machine language of the virtual machine in question. The high-level language virtual machine level provides the interface closest to the machine not requiring knowledge of system implementation and configuration. The main function of the model is heuristic - to provide a framework for determining which research directions may be useful to pursue.

The appropriateness of the model is examined through selected tests in Chapter 6. Tests are performed to measure the relative cost of processor operations and memory references in different environments. A memory hierarchy environment is simulated to provide a basis for determining the applicability of the model. The model is applied to the improvement of a heuristic for the Traveling Salesman problem. Results are presented to illustrate some of the basic uncertainties that must be dealt with in performance prediction in systems with memory hierarchies.

The suitability of the approach and a discussion of its applicability, along with suggestions for enhancements and topics deserving further research, are found in Chapter 7. The model is found to be simple to apply, analytically tractable for the simple test programs, and partially successful in predicting performance. Results show that the model is too general for accurately predicting the cost of memory accesses. Further research should concentrate on the effect on performance of various memory hierarchy configurations and implementations, particularly in terms of the relative sizes of hierarchy components and the effects of multi-programming.

Chapter 2

Program Performance

"...We have bad misconceptions about the possible exchanges between time and memory, tradeoffs between time and program complexity." [Min70, p. 198]

2.1. Introduction

Algorithm and program performance are the topic of much activity in computing science. Algorithms claiming improvements in the solutions to standard problems are common fare in journals. These claims may be based on theoretical analysis or on experimental results. Complexity analysis ignores (by design) implementation considerations. Implementation details, such as those concerned with memory hierarchies, affect the performance of high-level programs, and must be dealt with in a performance model for such programs.

2.2. Complexity Analysis

The behaviour of algorithms can be characterized in many ways, the most important of which deal with the time required to execute an algorithm as a function of the number of inputs, and the amount of memory (space) needed for the execution. Complexity analysis determines the time and/or space, in terms of elementary operations and the number and sizes of the data structures, needed to execute the algorithm. In *a priori* analysis, the complexity of the algorithm is expressed as a function of the input length. In *a posteriori* analysis, statistics are collected during the execution of the implemented algorithm when executed with some initial data, on the frequency of different operations, and on the time to execute each operation.

The improvement of existing and development of new algorithms are the subject of considerable research effort. This is done either with the practical goal of solving problems faster by computer, or simply with the desire to reduce the theoretical complexity (difficulty) of the solution. Both goals require a model of a computing device for the execution of algorithms and the definition of what is meant by the basic steps in a computation [AHU74, p. 4]. Complexity theory provides the theoretical background for

computation in an abstract environment. The most important motivation for formal models of computation is the desire to discover the inherent computational difficulty of various problems.

2.2.1. Computational Models for Complexity Analysis

It is important here to distinguish between computational complexity analysis, the study of what makes functions intrinsically difficult to compute [HoS78, p. viii], and concrete complexity analysis. Two products of computational complexity analysis have been the development of algorithms with the lowest (theoretical) asymptotic computation time and assertions concerning the minimum number of operations required to compute a given function. Unfortunately, many of the "best" algorithms from an asymptotic point of view, are extremely difficult to program, and require an amount of overhead that severely limits their practical value.

Concrete complexity analysis involves the development and exploitation of data structures and algorithmic methods in order to improve existing algorithms or to conceive new ones [Har87, p.145]. The distinction between the two approaches can also be described in terms of the complexity of problems as opposed to the complexity of algorithms. The notion of computational effort is of course defined differently in every model of computation. The peril of the situation lies in the failure by practitioners and teachers of computing to distinguish between the theoretical foundations of the two approaches, and as a result, to make unwarranted claims about the applicability of the models. The popularity of Heapsort is an example of the result of this confusion.

Initially, research on algorithms suffered from the lack of an abstract model on which to base its analysis. The effects of implementation were impossible to separate from the characteristics of the algorithm itself. It was difficult, if not impossible, to factor out the effects of computer performance, programming language characteristics, programming skills, and other such implementation details. Since changes in implementation detail affect running times, general conclusions based on this type of experimental analysis were not necessarily justified.

This lack of a common reference point motivated the development of an abstract model of computing based on the Turing Machine and the measurement of algorithm complexity by the number of operations needed for its execution [AHU74, Tar87]. The efficiency of an algorithm is measured not by programming it and running it on a computer, but by performing a mathematical analysis that gives asymptotic bounds on its potential use of time and space. The complexity is thus independent of programming details, the language in which it is written, and the specific computer on which the algorithm is implemented. It was hoped that conclusions derived from such an analysis would be of broad applicability.

Owing to problems in estimating input distribution, the worst-case analysis over all inputs of a given size was adopted as the measure of complexity. Constant factors in the running time were ignored so that the measure would be independent of any machine model and of implementation details. The measure tended to be analytically tractable, an important consideration when evaluating sophisticated algorithms.

It is, of course, essential that the model accurately reflect the costs of computation. The RAM machine [AHU74, p. 5], provides the model for asymptotic complexity analysis. The abstraction of the RAM machine concentrates on operations on data, and ignores instructions that are considered to represent at most a constant fraction of the cost of any efficient algorithm. Computational complexity operates in a framework of a one-parameter domain of computations, where the single parameter N is a measure of computation size. This accounts in some measure for the disparity between prediction and experimental results. By no means the only feature that determines the cost of running a program are accounted for by the arithmetic operations in the algorithm. Other considerations such as storage requirements and structural features in the computing system may lead to very different formulations of complexity.

2.3. Puzzling Evidence

The following examples add weight to the contention that the complexity of programs is a direct reflection of the complexity of the execution environment, and that no simple model can account for the performance of programs.

2.3.1. The Case of The Wandering Salesman

The six implementations of the Nearest Neighbour heuristic (see section 6.4) were executed on different processors (of the same type), at different times. This case study was intended to be another application of the model, but the results proved to be rather difficult to explain. It is included here as an example of the complexity of the environment as reflected in the performance of programs, and as an illustration that seemingly random behaviour may still show signs of pattern (see Fig. 2.1).

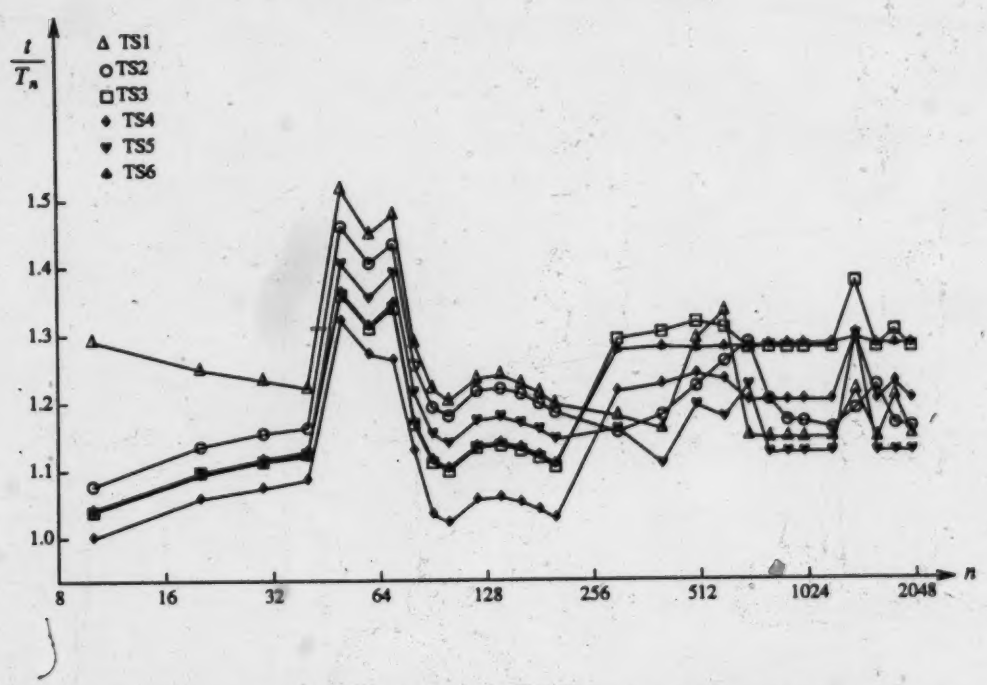


Figure 2.1 Unpredictability of Behaviour: Traveling Salesman Problem

Of particular interest are the matching anomalies in the behaviour evident for values of N between 40 and 200, and the highly irregular behaviour for values of N greater than 200. Any ordering of performance among the programs for lower values of N becomes undetectable for these higher values.

The partial ordering of results for lower values, and the coincidence of anomalies again suggests that some uniformity is to be found in the underlying memory management and file handling implementations. The hardware hashing function for cache (re)placement is perhaps responsible for the dramatic fluctuations in performance. A combination of space allocation in the cache and the mapping of memory contents to cache locations produces what appears to be a form of thrashing. The highly anomalous behaviour (compared with that displayed in Fig. 6.5, for instance), indicates an intractability directly related to the added complexity of the implementation of the memory hierarchy.

2.3.2. Two Wrongs Don't Make a Right

A second example of the intractability of real computing behaviour is illustrated by the case of the transitive closure algorithm. The first implementation uses addition and multiplication operations:

```
for k := 1 to n do
    for i := 1 to n do
        for j := 1 to n do
            A[i,j] := A[i,j] + A[i,k] * A[k,j];
```

The second implementation uses logical operations:

```
for k := 1 to n do
    for i := 1 to n do
        for j := 1 to n do
            A[i,j] := A[i,j] ∪ ( A[i,k] ∩ A[k,j] );
```

The uniform cost model of computational complexity suggests that the complexity of both algorithms is $O(n^3)$. Applying the logarithmic cost model results in a complexity of of $O(n^6)$ for the first algorithm

because of the size (number of bits) of the indices. The complexity of the second version does not change under this model. The real cost of executing the first implementation is closer to $O(n^6)$, and that of the second, $O(n^3)$. The cost is a reflection of the problems of dealing with arithmetic overflow in integer operations. If correct results are required, some form of multiple precision arithmetic will have to be implemented for the first case.

The important factor here is that it is not by any means clear which cost model is applicable in which case. Executing the two algorithms suggests that the logarithmic model is appropriate for the first case, and the uniform model for the second. Without reference to the practice of computing, however, there is nothing in the model, or in the algorithms, to suggest the proper mode of analysis.

2.4. Mapping Computations to Processes

Both cases suggest that blind acceptance of general models for predicting the performance of algorithms may lead to unpredictable, and possibly unpleasant results. The case of the wandering salesman suggests that complexities of implementation of "virtual" memory may lead to performance that reflects factors in the environment rather than qualities of the algorithm. The transitive closure case suggests that the abstraction represented by algorithms may be in fact an unwarranted simplification of the actual computational process.

The pitfalls of assuming correspondences between the structures and operations in algorithms and those actually executed may be described in the following manner. A computation C is first specified at the logical level as an algorithm. The mapping M_0 implements the algorithm as a program. The specification of implementation detail means that this mapping is from a virtual logical level to a real logical level. The next mapping, M_1, moves the implementation to the virtual physical level of machine language. One more mapping, M_2, is required to move to the real physical level of actual execution (see Fig 2.2).

We are interested in

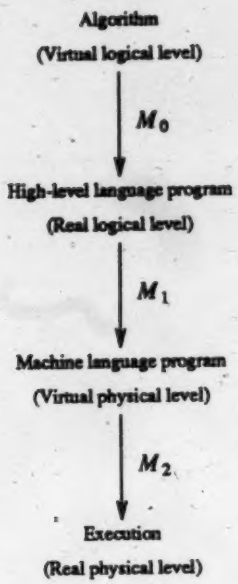


Figure 2.2 Mapping Algorithms to Executable Code

Time ($M_2(M_1(M_0(C)))$).

We can easily inspect C, $M_0(C)$, and possibly $M_1(M_0(C))$, but we can only *observe* the outcome of $M_2(M_1(M_0(C)))$. In addition, none of the mappings $M_0 - M_2$ has to be linear. As the Transitive Closure suggests, the move from the abstract multiplication to multiple precision arithmetic could be done as part of any mapping.

2.5. Conclusions

The RAM, or RASP machine formulation is not particularly suitable as a model for determining the execution costs of programs. Although execution of programs occurs on a high level language virtual machine that is an abstraction of the "real" computer, operations other than arithmetic operations are not so easily ignored. Computer programs may be characterized as computing a function from some finite range of possible inputs to a finite domain of possible outputs. This represents a change of state in the memory

space associated with the computation. There is some minimum time in which this change of state (reading and writing) can be accomplished, particularly in memory hierarchies. The computer is, of course, a physically embodied machine and, as such, cannot violate natural laws, particularly the physically determined bounds on the electrical and mechanical events within the computer hardware. In addition, hardware developments have changed the balance between the cost of processing operations and memory accesses, with memory costs now dominating [HoR77, p. 417-419, Mat86].

The emphasis on order and rate of growth provides a means of classifying algorithms, but little guidance in the development of efficient programs. Improvements within a given order may be significant. Complexity analysis concentrates on the algorithmic expression of the solution to the total exclusion of implementation considerations. The constant factor cannot be ignored when determining the cost of processing an input of definite dimensions. The constant may be the important part of the order equation when comparing different implementations in complex computing environments that bear little resemblance to the RAM and RASP machines on which asymptotic complexity analysis is based.

The changes in the non-arithmetic structure of the algorithm may have significant impact on performance, whereas the impact of changes in the arithmetic structure may be insignificant. The form of the cost functions may be predicted correctly by asymptotic complexity analysis, but the relative magnitudes, the constants, may be incorrect [Gen73, p. 104 ff. offers examples].

An algorithm's complexity is unaffected by which of the various isomorphic problems it is applied to, while the cost of the implementation is not. Measures of relative efficiency are seldom invariant over all problems of a given size. The abstraction in the algorithm is introduced explicitly to make it transparent to the various features on which the real costs depend: program structure, data representation, data management, and other language and machine implementation details. Of course, no abstract model can reasonably be expected to treat exactly the specific situation of computation. This would require a model so elaborate that it is more than likely to be mathematically intractable [Gen73, p. 107-108].

Computation time cannot be measured accurately by counting the number of elementary comparisons, fetches, stores and exchanges. Among other things, hardware techniques bias certain specific operations to run many times faster than other elementary operations. Instruction stacks, pipelined execution, caches, interleaved storage, and other such features a greatly improve the performance of programs. The analysis of the effect of such hardware features an be extremely difficult. As well, unusual instruction sets can introduce biases in program execution time that counter results predicted by worst-case asymptotic complexity analysis [Sto78, p. 138].

It may also be argued strictly on theoretical grounds that the asymptotic complexity model is inappropriate for the determination of cost. Cost deals with the use of resources. If a resource is unlimited, then no cost can be associated with. In this sense, the memory of the RASP machine is an unbounded system; such systems cannot be used to determine cost [Kli69, p. 70].

Chapter 3

Performance in Systems with Memory Hierarchies

3.1. Introduction

Presently, and for the foreseeable future, considerations of economic and technical nature make the large amounts of primary memory required by many applications infeasible. This need has been filled by memory hierarchies and their implementation, virtual memory. The major difference between programs executing in memory hierarchies compared to one executing in a simple memory system is not a logical one, but one of physical realization. A simple memory system, in which logical and physical memory coincide, does not incur most of the memory management overhead associated with bringing information in and out of primary memory (executable store).

3.2. Memory Hierarchies

The logical address space of an executing program may or may not reside in its entirety in primary physical memory. The portions not in physical primary memory are stored in a hierarchy of memory devices such as disk and/or drums. A reference to a logical address not in primary physical memory generates external intervention, which brings the contents of the address into primary physical memory from within the hierarchy, and establishes a mapping between physical and logical addresses. This may involve replacement, by which information in primary memory not presently required is returned to lower levels of the hierarchy. The implementation of memory hierarchies is generally referred to as virtual memory.

Memory hierarchies, regardless of their physical implementation and distribution of different technologies in the hierarchy, may logically be considered two-level memories because of developments in the top level. Caches are small[1], fast memories placed between the processor and main memory to reduce the

[1] The fact that 65K caches are becoming more common, and that 256K caches are not far away, simplifies somewhat the analysis of memory hierarchies. Hierarchies with large caches can be considered to have two levels: the cache, and main memory, with the higher levels being simply extensions of main memory.

amount of time spent waiting on memory accesses [Smi82]. Processors may operate at cache speeds while the economic advantages of a slower main memory are maintained. Blocks of recently (depending on the fetch and store algorithms used) referenced information are stored in the cache because future references are likely to be nearby. The overhead of accessing main storage is eliminated if memory references can be satisfied by the cache.

In contrast to the other memory hierarchy components, the cache is managed by hardware algorithms, deals with smaller blocks (32 bytes, for instance rather than 512), provides a smaller ratio of memory access times (5:1 rather than 1000:1), and, because of the last factor, holds the processor idle while blocks of data are being transferred from main memory to cache rather than switching to another task [KaW73, p. 31].

Cache-based systems are cost-effective to the extent that programs display locality and reuse data and instructions. The traffic between processor and cache is much heavier than between cache and main memory. This means that a large amount of information can be stored at main memory cost but mostly accessed at cache speeds. The cache also serves a buffering function between the processor and main memory. It reduces traffic, particularly if main memory is accessed by other subsystems such as input/output.

3.3. Effects on Performance

The effectiveness of memory hierarchies stems from the phenomena of spatial and temporal locality - addresses referenced in execution tend to be in close spatial proximity with addresses recently referenced, and this subset tends to change slowly over time [Den70]. As a result, the space needed (for code and data) to execute a program tends to be significantly smaller than that occupied by the entire program.

The introduction of memory hierarchies brought with it a new awareness of the effect of system configuration on the efficiency of programs. The implementation of virtual memory proved to be not necessarily transparent to programs running in such systems. This realization was triggered by the poor

performance in systems using paged memory of some programs designed for conventional systems [McC69, p. 164]. Programmers, under the illusion that memory now was unlimited, traded time for space. As a result, large and carelessly organized programs could generate excessive overhead in memory management. Many paged systems also suffered from severe fragmentation because storage requests are rounded up to the nearest integral number of pages [Den70, p. 156]. Poor page replacement policies cause unnecessary overhead, as do allocation policies that may lead to thrashing when memory is overcommitted.

There are a number of sources of degradation of performance in memory hierarchies. Some are inherent in the configuration of the system and its management policies. Page size and the speed of I/O devices are examples of these. Thrashing, for instance, is the result of a mismatch between the size of the locality and the amount of memory allocated to the program. Memory management policies themselves have a significant impact on the performance of a virtual memory system [Lau82, p. 10]. Other effects result from the interaction between memory management policy and the referencing behaviour of the program. The effects of memory hierarchies on performance has been investigated in a number of papers [AEH84, BrG68, Goo83, Ver86].

3.4. Program Transformation

Experimentation with performance in memory hierarchies was motivated by the desire to determine the causes for the overhead and whether it was in an acceptable range, or too high to outweigh the programming advantage inherent in the simpler approach (no overlays, etc.). The approach used was to analyze the causes of paging and to develop a set of techniques to be used during design and coding [BrG68, p. 483]. Two approaches prevail among these techniques. One requires that the programmer choose the best algorithm and implementation with respect to the memory management policy. This requires that the programmer understands clearly how a program affects the memory management policy - not a trivial task. The code generated may not be easily readable and maintainable, and any changes in the system might also require changes in the implementation of the program [Lau82, p.10-11]. The second approach uses infor-

mation about the referencing behaviour of a program to reorganize the virtual space to obtain better locality. The restructuring generally is based on connectivity graphs provided by either static or dynamic analysis of the program.

Program performance depends on the distribution of code and data into pages to improve locality and obtain small, stable working sets [Den70]. The general guidelines are designed to achieve locality of reference. This is a subjective definition. It means that a program's address space references (instructions and data) should be confined to as few pages as possible, for as long as possible. This involves organizing modules of a given size in such a way that interpage transitions are minimized. This will keep the working set small and stable.

Program transformation techniques are experimental [HaG71, p. 168], and can be characterized as packaging, the process of locating procedures and data areas to ensure good reference patterns, and algorithmic, controlling the order in which data areas are used or procedures executed. Three considerations are dominant in these discussions. Data should be referenced in a sequential (i.e. linear) fashion whenever possible. As much information as possible should be used from a given page in the same logical interval. Modules should be designed to allow them to be reordered during linkage editing to create better reference patterns. Orders of magnitude improvement can thus be made in the number of page faults and page residence times [Com67, HaG71, p. 168, McC69, p. 164].

The consensus is that careful attention to algorithm organization and program structure can improve the performance of programs in virtual memory environments. Experiments have shown that locality, and therefore paging behaviour, is strongly a function of programming style, and that it is possible to make significant improvements in performance by minor alterations in programming strategy based on minimal knowledge of the paging environment [BrG68, Say69]. It is not clear, however, that such techniques may be applied without knowledge of machine details (page size, memory size, and the like) [KuR68]. In addition, neither memory availability nor program behaviour are sufficiently predictable that a static approach to transformation can provide a reasonable solution. that may be applied during design and/or coding. They

represent an art, rather than a science [Rog75, p. 390].

3.5. Considerations in Modeling Computing Environments

The techniques described above all require more-or-less detailed knowledge of the environment in which the program is to execute. Many of the techniques are system- dependent. Rules for efficient programming for virtual memory run the additional risk of being configuration-dependent. On the other hand, the asymptotic complexity model is devoid, intentionally, of any consideration of implementation details. There appears to be no general methodology that is appropriate for the task of guiding the programmer in developing efficient implementations of algorithms. Ad hoc programming rules are not based on a general model of computing that can be examined or tested. The experimental method of determining cost has no general application because the domain is too big (machines, programs, operating systems, etc.) to allow the formation of a general model. The problem appears to lie with attaining the proper level of abstraction in the face of complexity.

The performance of an executing program is influenced by a large number of factors [Svo76, pp. 20-21]. The efficiency of the program itself is determined by such factors as the efficiency of the algorithm, the data structures used, the implementation language, and the programming style. Even the most (theoretically) efficient program must then contend with the computing system, the performance of which is a function of the system configuration, the operating system management policies, the efficiency of the system programs, the effectiveness of the instruction set processor, and the speed of the hardware components.

Much of the problem originates with what has been called the 'semantic gap' [Mye78, p. 11]. This concept represents a measure of the difference between the concepts of high-level languages and those of computer architecture. The objects and operations provided in computer architectures are rarely closely related to the objects and operations provided in the programming languages. This gap contributes to software unreliability, performance problems, excessive program size (and resultant performance degradation), compiler complexity, and distortions of the programming languages [Mye78, pp. 14-19]. Compilers

have not progressed to the point where they can bridge this gap to produce the best possible, or even good, assembly language translations of high-level language programs. [Sed78, p. 855, for example]. In addition, designers of compilers usually have comparatively little information about the way programming languages are actually used by programmers.

3.6. Conclusions

No suitable means of representing computing environments exists for the analysis of high-level language programs. Complexity analysis offers the generality of the abstract machine model, but discards the implementation details that distinguish programs from algorithms. Its model is *abstract* rather than general. It is based on a general model of computation, but this model is itself an abstraction with no reference to implementation details. The model offers guidelines for program evaluation and improvement, but evidence suggests that this advice is of a dubious nature. Programs tend to deal with finite inputs of average (as opposed to worst-case) data, and performance analysis should reflect this. In addition, implementation details appear to have more effect on the behaviour of programs than is allowed by the RAM model.

Chapter 4

Description and Analysis of Computing Systems

> Simplicity is not to be found at the beginning of inquiry, but, if at all, only at its end [Ack64, p. 53].

4.1. Introduction

Multi-programming, virtual memory computing environments exhibit a high degree of non-determinism at the macro level. The mediation of such processes as schedulers and memory managers produces a number of possible machine states that precludes any attempt at meaningful analysis at a detailed level. The programs performing the scheduling, resource allocation, and memory management policies use heuristics that aim at maximum utilization of the resources in the environment. The scheduling and memory management systems (the computing environment) and user and system processes interact to form a complex whole. Knowledge of the organization and behaviour of individual components is not able to provide a general understanding of the functioning of the complex whole.

At a micro level state transitions are perfectly deterministic given a detailed knowledge of the algorithms employed. The state transition function and output function uniquely determine the new state and output from a specification of the initial state and subsequent inputs. At the macro level, however, the situation is more difficult. The system demonstrates *equifinality;* the results of computations are not altered, and the workload is managed just the same. The exact sequence of operations, allocation of resources, etc. in accomplishing this is not predictable or repeatable.[2]

Rather than conforming to the *mechanical ideal-type*, the dynamics of the computing environment exhibit opportunistic or strategic behaviour in response to local parameter and workload changes. The dynamic forces are likely to be transparent rather than tangible or manipulable. Heuristics are special

[2] Quantum mechanics is faced with the same problem. The universe with which it deals is inherently stochastic. The updating of states is still a deterministic process, but the state no longer gives precise predictions, only allowing determination of the probabilities of various outcomes [Arb72, p. 62].

problem-solving techniques used to control the combinatorial explosion of possibilities in the solution space. They use knowledge about the problem, about the organization of the solution, and about the goals. The cost of this reduced complexity is the absence of completeness in the solution space. This means that future-state conditions cannot be accurately or completely inferred from a starting-state or from historical conditions.

This chapter examines ways to describe and analyze computing systems. The approach chosen is the reduction of the complexity of integrated hardware/software/firmware computing systems to organized collections of logical resources. The tools we use are those of systems science, focusing on the boundary, structural, and dynamic dimensions of systems and their component subsystems.

Computing systems are "well-defined" because the design process decomposes the system into component subsystems with specific, clearly detailed functions, and clearly defined hierarchy and interactions between system and subsystems. Interactions and relationships between components are clearly specified. Based on this type of analysis, it is determined that the virtual memory, multi-programming environment is a complex system because it exhibits adaptive behaviour, and the subsystems interact in a complex manner.

In particular, computing systems are described as hierarchical systems because of the extensive use of the concept of virtual machines in their design and use. A virtual machine constitutes a complete logical description of the computer at a particular level of abstraction. High-level language programs are written for a virtual machine defined by the language.

4.1.1. Analysis of Complex Systems

The mechanical aspects of computing systems do not easily reveal their nature to reductionist approaches. Although, in the final analysis, the operations of the machine can be described in terms of the principles of electronics, one is unlikely to make much sense of a computer *system* by appealing to electronics alone. Similarly, the dynamic aspect of the computing environment is not analytically tractable in all its detail. Investigating queueing disciplines, page replacement policies, etc. will provide little illumination for the computer scientist interested in the *systems* aspects of computer system behaviour. The empirical-inductive analytical modality is not likely to provide an accurate analogy of the system as an operating entity.

In treating systems that are intractable to the empirical-inductive modality, we are forced to adopt an approach that does not depend for its significance on the availability of a high-quality data base of experimental results. The hypothetico-deductive modality is characterized by the generation of hypotheses whose origins are not derived exclusively from observation. The method allows us to manipulate concepts, which do not take the form of dogma or categorical imperatives, but structures that may serve as guides and bases for investigation *and whose ultimate validity or rectitude depends on the extent to which they are subsequently empirically validated* [Sut75, p. 88]. We thus approach the analysis of complex systems from two perspectives, each with its own authority. The observations cannot be denied, and the fundamentals that inspired the investigation cannot be disputed.

4.2. The Systems Approach

It is the organization of the components, and not their physical properties, that largely determines the behaviour of the computing system. This organization reflects the model of computation inherent in the design of the computer. Two models of computation always exist: one at the logical level that describes what the system does or is required to do, and one at the physical level that describes how the logical components are to be allocated or realized in terms of some combination of hardware and software. The system

algorithm, or what the system is supposed to do, is separated from the hardware by one or more logical layers that may be referred to as virtual machines. The function of these virtual machines is to provide an interface between the logical and physical models. This approach may be extended to a generalized logical view in which no assumptions are made regarding the hardware or software nature of the concentric virtual machines [BoB85, p. 50].

The systems science approach holds certain appeal and promise as an analytical framework for the study of complex phenomena. Its principal strength lies in its insistence on conceptualizing interacting events as a totality or unified whole. Systems analysis seeks to identify relations among parameters that influence behaviour, and to construct models in which that influence can be ascertained. Starting with a model which identifies system parameters, it is possible to study the influence of these parameters and their mutual effects on performance. The behaviour of many systems of practical interest, which may be too complex to be studied in all manifestations of behaviour, can often be investigated using certain simplifying assumptions based on prior knowledge of the structure or organization of the system.

The individual steps in the technique are not new or unique. The principal contribution of the method is in the recognition of purposive behaviour by a system, and the use of formalized procedures and explicit analysis, suited to the system being studied, and applied according to established rules.

4.2.1. Analysis and Description of Systems

The system scientist studies "objects" by applying abstract ideal types against which all real-world systems may be compared - constructs that serve as "masks" we can wear to bring some tentative, *a priori* order and structure to virtually all segments and entities of the world we live in. These ideal-types should not only, in a sense, mirror reality, but they should also serve to suggest what instruments and techniques are appropriate for the analysis of systems approximating one or another of the ideal-types [Sut75, p. 98].

The formal definition of a system should isolate the properties that serve to distinguish proper *systems* from other real-world phenomena. A proper system should meet the following criteria [Sut75, p. 99]:

[1] A system constitutes an entity of determinable morphology. It must be in, or capable of obtaining, a state of sufficient integration to separate it from its environment.

[2] It must contain or encompass two or more morphologically determinable entities. That is, it must contain multiple, differentiable subsystems.

[3] A system must be capable of, or actually exercising, constrained animation among its subsystems, such that their behaviour is not entirely autonomous. At least a portion of the energy potentially available to subsystems must be used by the system for the "larger" purpose or for maintenance of the integrity of the whole.

The three *dimensions* of a system are based on these criteria. The criterion of integrity is met by the *ecological dimension*. This encompasses the "without" of systems, and is concerned with the external configuration, the interface with the environment, and relations with other systems that may affect it, or be affected by it. This dimension really considers the interface properties of the subject system. Also known as the boundary dimension, it describes the functional limits of the system and its interactions with its environment. For computer systems, we can also talk about system *architecture*, the detailed and complete specification of the user interface at each level of description.

The *domain dimension* contains the system's substantive properties. The domain deals with the structural aspects, treating the resources and the broad, static patterns of internal behaviour that might be considered prevalent. It should describe the components of the system, and the relationships among them.

The *dynamic dimension* is concerned with the nonstatic, processual properties of the system. This includes the work done by the system, alterations in the patterns of interaction among components or between the components and the system as an integrated whole. These behavioural properties describe which components of the system can potentially interact with which others, and under which conditions and constraints.

4.3. Computer Systems Design

In a large part the behaviour of computers is governed by simple general laws, and what appears as complexity in computer programs is, to a considerable extent, complexity of the environment in which the program was designed to operate. From the outset of computing, the design and structure of software have been determined by the physical limitations and constraints of the hardware, coupled with the desire to maximize use of these hardware resources.

Most computers have certain common organizational features based on the von Neumann model of computation. They can be decomposed into an active processor and a memory in combination with input and output devices. They are all capable of storing symbols (programs) that can be interpreted by a control component (executed). Symbols generally have to be moved from the larger memory components into the central processor before they can be manipulated. The systems are capable of only simple basic actions: recoding symbols, copying symbols, moving symbols, erasing symbols, and comparing symbols. Insofar as their behaviour reflects broadly the broad functional characteristics we have described, and is independent of the details of the hardware, we can build a general empirical theory.

The process of systems design moves from statement of requirements and constraints to a definition of the system that meets the requirements and constraints. Various design methodologies, which are defined as algorithms executed by the designer, are used to deduce the structure and constituents of a system that satisfies the requirements. The specifications are derived from some form of analysis of the original problem that is to be solved. The final system design of a computer details the hardware and the software and how they operate together to perform the required processing. Design requirements are based on a model of computation.

The most general model is that of the computer as computing utility. No special purpose is attributed to the computer: it is a source of more or less mysterious control and computational power. Purpose is specified in the programs executed. The machine is defined by the programs in execution at any one time [Dor79, p. 7]. This view developed with the concept of time-sharing, giving users the impression of using a

dedicated machines. (Micro-computers have, of course, contributed to the prevalence of this view).

Two models of the system always exist [BoB85, p. 50]. The logical system model is a complete description of what is to be implemented. The logical design process begins with a set of requirements and organizes these into a logical description of the data-flow and control features. The first concern here is with logical correctness. The process can be described as the reverse of architectural system analysis. Analysis begins with a physical system and breaks out the data-flow and control features such that the structure and operation of the system can be examined. The result of this design phase is the creation of a hierarchy of virtual machines that can ultimately be realized as some combination of hardware and software.

4.4. Virtual Machines

The key concept in a system perspective on computing systems is that of the virtual machine. It is an abstract interface definition, representing the visible system features at that particular level of system description. The kernel, for instance, is a virtual machine comprising a special critical region of software and its supporting hardware, that supports the basic operations of a computing system. It is a combination of hardware and software that provides a level of interfacing higher than the basic hardware itself. High performance over many simultaneous domains may be guaranteed by scheduling virtual machines onto physical machines. Otherwise, hardware (and software) resources will most likely not be effectively utilized.

Virtual machines are a central concept in the design methodology. Much of the power of computers is achieved through multiple maskings, the creation of virtual realities. Levels of symbols, each self-contained in its internally self-consistent logic, are built on top of each other. Each level simplifies the details of the implementation of the level below it that supports it. The power derives from the leverage exerted by commands at higher levels over the operation of the machine at the lower levels. This leverage extends to the domain of design, where abstraction allows the manipulation of logical entities without the

need to know details of the implementation of the entities.

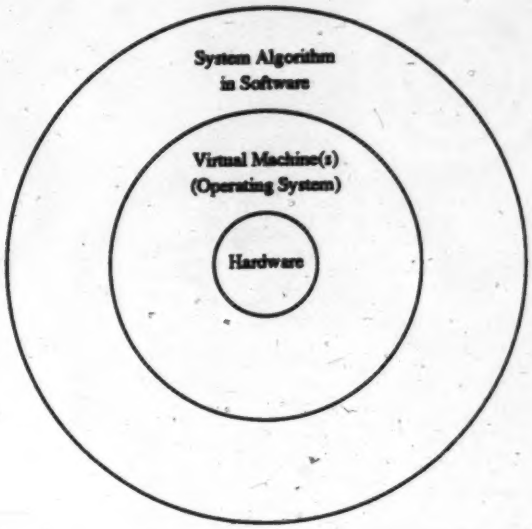


Figure 4.1 Logical system view

At the centre of the logical system view is the hardware, unspecified, on which the system is to be implemented (see Fig 4.1). The outside layer specifies the required processing. The two are separated by one or more logical layers that are referred to as virtual machines. Virtual machines act as logical interfaces, providing a logical view of the system to the next higher layer. This logical view need not correspond to the physical view held by the hardware designers. The function of virtual machines is primarily to make the implementation transparent to the user.

4.5. Virtual Machines and Instruction Sets

The functional behaviour of the components of one computer is similar to that of any the components of any other computer, regardless of the design of the computer. The major differences between computers lie in the way the modules are connected together, the performance characteristics of the modules, and the way the computer system is controlled by its system software. Architecture is generally interpreted to mean the study of the organization and interconnection of components of computer systems [Mat80, p. 3]. This thesis considers it in the more general definition as the interface presented to the user. There may thus be several versions of one architecture depending on the interface presented to the user - from the micro-programming level that matches closely the physical detail and structure of the hardware, to the high-level language level that sees a high-level virtual machine implementation of the logical design.

There is considerable interdependence between the architecture of processors and the structure of algorithms. The development of algorithms involves the representations of the required processing in terms of the available data-flow and control features of the processor; that is, the algorithm is written for the virtual machine defined by the instruction set.

The operational level corresponding to the architecture, as expressed in the instruction set, is straightforward. The instruction set can be thought of as a set of procedures, one for each machine instruction. The instruction set is essentially limited to operations that access and alter the contents of memory (or the accumulator or the program counter). The basic control mechanism is derived from the sequential fetch/decode cycle. The control task gets the next instruction at the memory address stored in the program counter. The instruction is decoded, and a procedure call executed to the instruction set package. Execution is discrete: the execution of each instruction completes before the next instruction is fetched.

A computer's primitive instructions form a language used to communicate with the computer. Each virtual machine has a machine language, consisting of all the instructions that the machine can execute. A machine thus defines a language, as a language defines a machine that can execute all programs written in the language. The fact that programmers need not be aware of how the level being used is implemented

leads to it being viewed as a real physical machine, even though it is only "virtual." This structuring enormously simplifies both producing complex machines, and writing programs for them [Tan76, p. 12]. There exists an absolute, elementary level of detail for which no further refinement of description is required. This level of detail for a virtual machine is determined at design time and specifies the operations and operands required at the interface, and thus the capabilities, of the machine.

A programming language in fact implements a virtual computer with its own set of data types (e.g. floating point) and operations (e.g. indexing) in terms of the real (or virtual) host computer with its own data types and operations. Higher-level languages provide facilities more suitable to the applications for which they are designed, and eliminate many lower-level (host machine implementation) details from programming [Mac87, p. 5]. Programming languages may thus be viewed as defining virtual computers that are in some respects better than the "real" computer on which they are designed to run.[3]

A programming language is implemented by constructing a translator that translates programs in a machine-independent language into programs capable of being executed by some computer, which usually is a virtual computer implemented partially in hardware and partially in software. The virtual computer on which the language is implemented is distinct from the language itself. Different implementations of the same language may be hosted by quite different (virtual) computers. The language implementor has wide latitude in the structures and algorithms used to simulate the virtual machine defined by the language

Although languages may differ in their details, they are often similar in their general structures. These general structures define a language framework, in which are embedded application-oriented datatypes, functions, and operators that optimize it for a particular class of uses [Mac87, p. 545]. A high-level language virtual machine level is thus defined by this general language framework, while accommodating variations in the implementations of the individual languages. This model agrees with the logical design approach of partitioning logical levels to provide for different modes of operation within the same logical

[3] This is a generalization. Consider what Dijkstra said about PL/I, a language so large as to be unmanageable and too complicated to be mastered by most programmers. He called it a "fatal disease," and referred to the "frightening size and complexity of the defining documentation." Quoted in Mac87, p. 176.

framework.

4.6. Computers as Complex Hierarchical Systems

A hierarchy, or hierarchic system, describes a system that is composed of interrelated subsystems, each of the latter being hierarchic in structure until some lowest level of elementary subsystem is reached [Cam82, p. 196-200]. The definition of elementary subsystems and the number of partitions are somewhat arbitrary, depending on the objectives of the analysis. The complex system is logically decomposable into subsystems that we may call virtual machines. Each virtual machine hides the complexity of the machine below it, and thus subordinates operations on the level below it. This falls within the definition of true hierarchic systems.

The various levels, or virtual machines, in a hierarchical system may come about by construction or refinement/abstraction [Das82, p. 98]. In the first case, a mechanism at level *i* is constructed on a mechanism at level *i - 1* if the objective is to provide some capabilities at level *i* that were absent at the lower level, or to suppress lower level capabilities. Alternatively, level *i* may be chosen to hide details of implementation on the lower levels, thus providing a higher-level capability. Bell and Newell [BeN71] remind us that these levels are not equivalent in the sense that anything said one way can be said another way. Each description level performs a function that cannot be performed adequately by another. Each level defines a machine's functional capabilities from a specific requirements viewpoint.

4.7. Conclusion

Computers can be described in terms of hierarchies of virtual machines that describe completely logical levels of implementation of the design. Hardware and software are logically equivalent with this kind of interface structure. The architecture of a computing system is defined as the interface as presented in the instruction set. The instruction set for the high-level language virtual machine is the general framework for the high-level languages used to program the machine.

Chapter 5

A Cost Model for Computing

5.1. Introduction

The complexity of computing systems is made manageable through systems analysis using the concept of hierarchically ordered logical layers of functionality. Each of these virtual machine layers presents an interface to the user that is completely specified in the language used to program that level. The architecture of the high-level language virtual machine level is defined by the general frame work of high-level languages. The cost of executing programs is examined in the framework of the architecture of this virtual machine level.

5.2. The Cost Model

The cost model for program execution is a descriptive model. It makes no claims about detailed operation of the system, but describes the general principles that determine the externally visible characteristics of the system. It is validated by description of the underlying system, and by data generated by a number of tests and simulations. Because the model is descriptive, it cannot be used directly for simulation. Instead, various aspects of the model are described, and test data relating to these aspects are examined.

5.2.1. Interface (Environment)

The interface dimension of a system describes the external configuration, the interface with the environment, and the relationships with other, external systems.

5.2.1.1. Computers as Complex Artificial Systems

Complexity is not an intrinsic property, but is dependent on our knowledge of things or processes [Mow76, p. 230]. Situations with no apparent structure, those with a large number of variables that appear to behave erratically or in an unknown manner, may still in their aggregate behaviour exhibit statistical regularities. Resulting ambiguities may be lessened through the processing of large quantities of information, or through the understanding of purpose in the object being studied. There is a danger here, as with all empirically perceived regularities, of seeing order where none exists.

Organization is a critical feature: a system with a large number of interacting components whose behaviour is structured and patterned may still appear intractable. It is likely, however, that the criteria for detecting structure or pattern is embedded in the purposive or goal-seeking behaviour demonstrated. Knowledge of the purpose of a system can thus reduce apparent complexity [Mow76, p. 230].

Computers can be described as artificial systems with known organization and observable behaviour. As such, they may be systematically described and analyzed. Artificial systems have properties that make them particularly susceptible to simulation via simplified models. Inherent in the concept of artificial systems is the notion of purpose [Sim69]. Design incorporates all feasible requirements associated with the purpose to be achieved. Decomposition into subsystems associates specific functions, structure, and operations with each component.[4] These are related to the overall system function.

The speed with which computers perform basic operations allows us to infer little about the physical components and their laws. Almost no interesting statements can be made about the operation of computers by the reductionist approach of describing the specifics of the hardware. As the complexity of a system increases, the ability to make precise and yet significant statements about its behaviour diminishes until a threshold is reached beyond which precision and relevance become almost mutually exclusive characteristics.[5] A computer is an organization of elementary functional components in which, to a high

[4] From the science of the engineer : "The distillation from the general statement of a problem of one or more precisely defined problems whose solutions can be found, which could either partially or completely answer the original problem." [WeA70, p. 7]

approximation, only the function performed by those components is relevant to the behaviour of the system as a whole. Almost the only properties that are detectable in its behaviour are the organizational properties related to its function, and the speed with which it performs basic operations does not allow us to infer much about physical components and their characteristics.

When analyzed in terms of the functionality of components, computers appear to fit the description of simple tangible boundaries that may be easily controlled or adjusted. Their components are arrayed in a neat, observable hierarchy with essentially deterministic relationships between them, and the behaviour of the subsystems is usually highly constrained.

Analytically, this should mean that computer systems are highly accessible to the observer, that their states and processes are easily quantified, and that they are easily manipulated (in lab situations). Given a set of starting-state conditions or a set of historical properties, the observer should be able to induce or predict future states with a high probability of accuracy.

The computing system is divided into logical levels of operation referred to as virtual machines. Each high-level language used to program the computer defines a virtual computer. Individual processes executing high-level language programs define virtual machines. These user processes interact with the underlying "real" machine through the environment or interface provided by the operating system. The "real" machine is a slave of the operating system, and its services are provided in an orderly fashion to the user virtual machines.

⇒ The environment in which programs execute is the state of the machine produced by the operating system in response to the characteristics of the computing work being performed. It is thus a function of the algorithms and data structures comprising the operating system and the effects of past, present, and anticipated requests for the resources managed by the operating system.

---

[5] A corollary principle may be stated as, "The closer one looks at a real-world problem, the fuzzier becomes its solution." [Zad74, p. 56]

Computers may be seen as integrated sets of algorithms and data structures capable of storing and executing programs [Pra75, p. 13,29]. They may be realized in hardware, as collections of physical devices, in firmware by micro-programming a suitable hardware computer, through software, simulated in some other programming language, or through some combination of these techniques as considered appropriate.

Multi-programming led to drastic changes in the way computing systems are viewed. Tasks are executed by a conceptual machine, supplied to the task by its initiating regime as part of its initial environment. As a task executes, so its environment, unlike a processor that is fixed, changes. The environment allocates the logical resources of the underlying hardware, software, and firmware to competing processes on a dynamic basis. In this environment, a process is represented by an execution record, and the conceptual machine is implemented by the operating system, which is a cooperating set of parallel tasks designed to deal directly with the hardware. An executing process (allocated a processor by the operating system) refers to virtual memory, a sort of symbolic memory, which is mapped at run-time into actual memory.

In the hierarchy of virtual computers, the operating system generally provides simulations for a number of primitives and structures not directly provided by the hardware. External file structures and file management primitives, for instance, make this the virtual machine usually available to the implementor of high-level languages. Along with the provision of these primitives, the implementations of other functions, such as those dealing with input/output, scheduling, and memory management, are hidden from the user.

We thus have programs, represented by tasks or processes, running on a machine, represented by the operating system, referring to memory locations that are represented in a virtual memory. Each of these may be represented as a complex system, and as subsystems interacting to form a larger complex system. This approach may yield a multileveled logical view of the system in which hardware extends to various logical levels within the system.

⇒ Performance is concerned only with virtual time, that is, elapsed time in the context of the execution

of a specific program only. In real-time, the resources of the "real" machine are shared with other virtual machines, both "user" and "system"; in virtual time, execution is uninterrupted.

The "gross" behaviour of the programs is defined in terms of user time, rather than elapsed time. This may be defined as time spent by the host in the execution of instructions in the program. This is a virtual measure of time, because the vagaries of scheduling are beyond the control of the user-as-programmer. This virtual time has a lower bound that is approached on a single-user machine, or in a multi-programming environment with few competing processes.

Competing processes define separate virtual machines under the control of the execution environment. Heuristics, whose goal is some measure of optimum resource use, determine the order of programs to be executed and how much of the program is executed in one "round." A program in execution interacts with its data through the memory management subsystem, and with other resources required through the processor. Each virtual machine is unaware of other virtual machines, of the details of its own implementation, and of the details of external activity on its behalf.

⇒ The user interface with the computing system is a high-level language virtual machine. The functionality of the machine, the organizing principle of its operation and capabilities for the user, is defined by the high-level language used to program the machine.

⇒ Program performance is measured at the interface with the virtual machine whose architecture is defined by the general framework common to high-level programming languages that defines the logical resources available to high-level languages.

The cost model for computing is a model of high-level language program execution. A model for the prediction of performance of programs must be stated in terms of primitive operations that (a), represent the required processing and (b), represent the cost of executing the instructions specified by the algorithm. A model concerned with performance of programs cannot be effective unless the user (as programmer) has

access to the descriptive variables (the sources of cost) in the model. These primitive operations in the programming language fall into a limited number of classes that share execution characteristics based on their implementation. The cost of execution can only be determined at the interface between the program in execution (as expression of the user problem solution) and the computing system as expressed in the environment provided for the execution of user programs.

The cost of program execution can be established by studying the behaviour of programs with known characteristics. Statistical correlations can be established between these known characteristics and the gross behaviour displayed in execution. These correlations may be anticipated and understood by reference to knowledge about the structure and operation of programs and computers.

The model of computing in which we are interested exhibits two dimensions of virtuality. On the vertical (hierarchical) dimension, we are examining a virtual machine that is several layers removed from the hard/firm/software that actually executes instructions. On the horizontal dimension, it may be one of a number of virtual machines sharing the same framework or model of computing based on the general characteristics of high-level languages. Each program defines a virtual machine with control flow and processing instructions, and local and external (indexed variables) memory. To these virtual machines, scheduling and memory management are completely transparent.

⇒ The instruction set of the high-level language virtual machine is completely represented by the high-level language used to program it. Algorithms are not equivalent to the programs that may implement them, because they do not specify all the details upon which execution depends.

High-level languages define virtual machines for the execution of programs written in that language. Programs are implementations of algorithms, which are general statements of solutions. Many different implementations of a single algorithm are possible, differing primarily in the organization of the tasks performed by the algorithm itself, and the choice of data structures and operations thereon. Programs in the same solution class (based on the same algorithm) can thus take many forms and, as a result, display widely

varying behaviour and performance. In addition, overhead and variation in implementation details are introduced in the form of compiler added work (e.g. instructions added to cope with the architecture) and operating system work (e.g. system calls required).

5.2.2. Structure

The structure of a system describes the system components, their place within the system, and the relationships among them.

5.2.2.1. The von Neumann Model

Specifications should not predetermine system structure. The reality of most design processes, however, is the existence of constraints. Constraints, or non-functional requirements, must be satisfied. Constraints occur in the form of existing technology or specific user requirements such as the need for backward compatibility, the need to integrate with existing systems, or financial considerations. The design of computing systems is particularly constrained by the dominance of the von Neumann computational model.

The physical implementation of the logical system design is based on the traditional von Neumann architecture model derived in the 1940s. It is common to talk today of a class of machines as von Neumann machines and to say that most current machines belong to this class. The computer was originally conceived as a machine for doing "arithmetic".[6] It was natural that its inventors should have visualized a program as a linear sequence. It is difficult to envisage any formal, explicit process of reasoning that is not modeled in this way.

The classic von Neumann processor architecture consists of four basic elements: the ALU, a control unit, a memory unit, and an input/output unit [BoB85, p. 275]. The first two functions (data manipulation

---

[6] John von Neumann: "In as much as the completed device will be a general purpose computing machine, it should contain main organs relating to arithmetic, memory storage, control and connection with the human operator." Quoted in [Dor79, p. 8].

and control) are both functions of the processor and may thus logically be treated as one. Because input/output is performed by peripheral devices and processors, it appears as a single CPU instruction and may be treated as such.

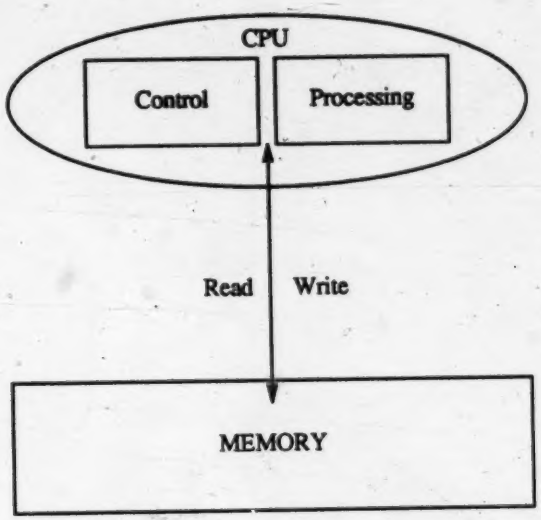


Figure 5.1 Logical Model - Controlled Manipulation of Memory

The logical basis of the von Neumann machine operation is the controlled operation of memory contents, one word at a time (Fig 5.1). The instruction set represents the user-visible data-flow and control functions of the machine. We have an architecture that is defined by a low-level machine language consisting of instructions that perform simple operations on elementary operands. Although developing technology has altered the appearance and improved the performance of successive generations of computers almost beyond recognition, the basic structure has remained unchanged.

⇒ The structure of programs is strongly related to that of the machine on which they execute.

Overall systems design specifies an algorithm for doing general computation. Each virtual machine defines and instantiates an aspect of this general model. The structure of each machine is reflected in the

structure of the language used to program the machine. The logical structure of the machine is that of the von Neumann machine.

Logically, programs must reflect the structure of the machine on which they are to be executed. Rather than speaking of algorithms, or programs, we can talk about "a method of solution for problem P on device M" [Kor66, p. 89]. This is defined as a description *in a language comprehensible to M of discrete steps performable by M* and an ordering of these steps such that, given proper data, if M performs the prescribed steps in the prescribed order, a solution to problem P will result, (if one exists). All detail required for the execution must thus be specified.

Physically, programs are sub-divided into modules, matching the decomposition of the machine into logical entities performing a specific function. A program, or module, has two kinds of data. "Local" variables represent short-term values, each one specifically named because it performs a specific function in the execution of the program. These variables generally are part of the control structure of the program. External data is that upon which the program operates. It consists of files or simple or compound indexed data structures. The size of external data represents the "size" of the problem.

⇒ The resources of the virtual machine are logical resources. The model of computation inherent in sequential computing imposes a logical structure on computing independent of implementation.

There are two dimensions to this "virtuality" of resources. Firstly, the resources of the host machine are provided in a manner that is transparent to the executing program. No knowledge, or statement of detail of implementation is required. Programming languages do not care how the machine resources are supplied, and consequently the virtual machines defined by the languages regard the resources as logical rather than physical. The program virtual machine performs two types of operations, based on the types of statements allowed in the language. Processing and control operations define the function of the logical processor. Data structures define the memory configuration of the virtual machine. Local memory is defined by the "register" variables, and indexed structures define an external memory. Secondly, the resources are

virtual in the sense that they represent collections of devices and concepts that are organized to represent one functional component of the system. Memory hierarchies and processors both represent this kind of complex functionality.

5.2.2.2. Memory Limitations

There is little argument about what kind of memory is needed for practical computers. The fundamental requirement can be reduced simply to large, random-access high-speed (both read and write) memory systems at low cost. The limited size of main memory, and the resulting limited address space provided to the user, has had its effects on computer architecture in general, and processor design in particular. Hardware and software systems for storage allocation, data management, buffering, paging, and so on reflect the compromise solutions involving trade-offs between cost, size and speed of the alternative memory technologies.

Virtual memory is not only the result of the limited size of main memory. Multi-programming requires that programs and data be easily and efficiently relocatable. Common sense also tells us that the major portions of the code and data of most programs are infrequently used in execution, and need therefore not be kept in main memory during execution of the rest of the program. Virtual memory is thus more than an inexpensive solution to the limited size of main memory; it is an essential component of a multi-programming computing environment.

The size of a problem that can be represented as and solved by a computer program has always depended on computer storage capacity [Mat86, p. 44]. Practically speaking, however, the storage capacity of a computer can be made nearly infinite by the use of disks, tape, etc. (see Fig. 5.2). Problems of speed in accessing this memory may make certain computations impractical; nevertheless, memory hierarchies mean that no fundamental limit on problem size (in terms of memory) exists.

Practical limits do exist. Most large problems require storage exceeding that of main memory. In this type of problem, the limiting factor turns out not to be storage capacity, but the speed of the CPU and

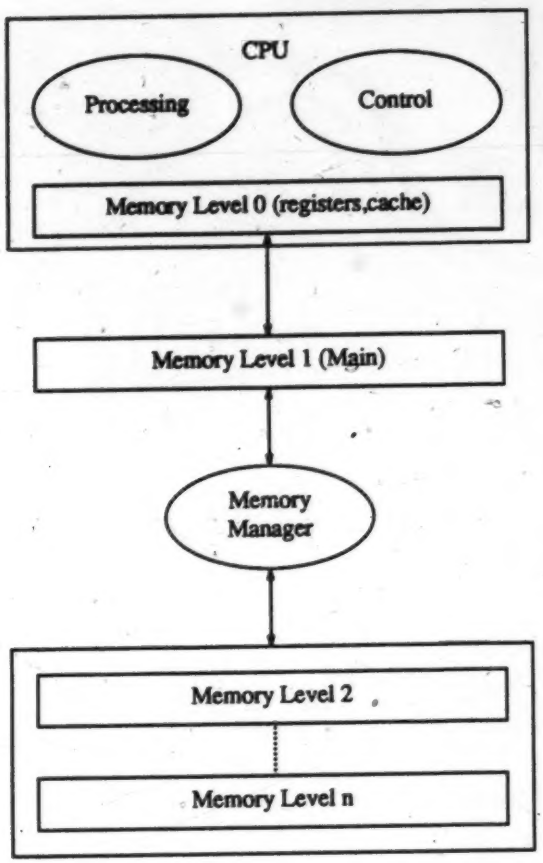


Figure 5.2 Von Neumann Model As Implemented

the access time and data rate of secondary storage. Some of these limits may be overcome in theory, but not in practice, if the cost of overcoming them is unreasonable. Since all operational computing systems use physical devices, the laws of physics are an obvious inherent limitation. The speed of signal (information) transmission imposes a lower bound on movement of data required for operations. Other limits are imposed by the need to maintain comprehensibility and minimize complexity [Gra75, p. 401-2].

5.2.3. Dynamics

The dynamic dimension describes the work done by the system, and patterns of interaction among components and between components and the system as a whole. We are concerned here with the operation of the high-level language virtual machine.

5.2.3.1. Algorithms and Execution in Abstract Environments

⇒ There is no directly discernible connection between operations in the machine that can be deemed to have costs associated with them and the operations specified in algorithms or high-level language programs.

Problem solutions may be described in a number of ways, on different logical levels. They may be represented as hierarchies, as in Fig. 5.3 [Fly77, p. 41-2]. *Algorithms* specify the *tasks* to be performed in the solution. The tasks are described in *higher level language statements*, each of which is represented by a *sequence of instructions*. These instructions cause *state transitions* in the host machine. Each level represents the same object (problem solution) in a different way; the solutions should be equivalent.

A direct relationship exists between the algorithm and the state transitions caused in the host (executing) machine by its implementation. The transformations, however, are not rigidly determined. An algorithm may choose from a number of tasks to accomplish its purpose. These tasks may be described in a number of different ways, in any number of languages. The machine on which execution occurs determines how the higher level language statements are interpreted into sequences of machine instructions, and how these affect the state of the machine.

This model of computation regards algorithms as being implemented as single, distinct, identifiable tasks. This makes it possible to consider the cost of executing an algorithm by determining the connection between the tasks specified in the algorithm and the state transitions effected in the host machine. This is

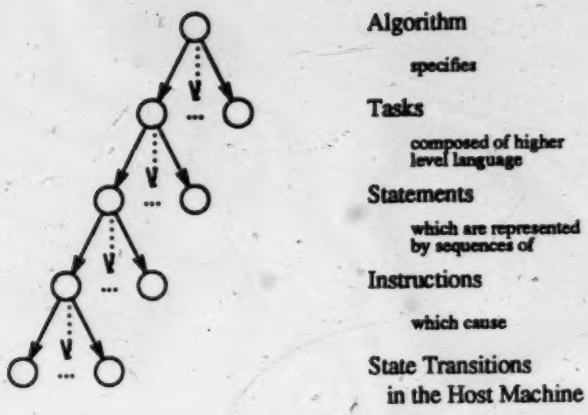


Figure 5.3 Hierarchical Structure of Computer Problem Solution

the approach used by asymptotic complexity analysis and its RAM machine.

Multi-programming machines, however, execute a number of tasks concurrently. That is, the workload executed by the host machine is not composed of identifiable programs, but of instructions from a number of programs. The instructions can neither be identified as belonging to a particular task, nor is this in any way required. In effect, the hardware is a slave of the environment in which programs execute, and such matters are of concern only to the environment (see Fig. 5.4).

If there is a relationship to be found between the steps specified in an algorithm and the state transitions in the host machine, the rules used by the environment to create and manage the host machine's workload must be examined if possible. If it is not possible to determine these rules, or if they should prove analytically intractable, then operations in the environment, and not in the host machine, must be examined to determine the cost in the execution of algorithms.

The execution of high-level language operations is done without reference to the context of the operation, and without a need for the executing hardware to know the owner and context. The operating system, through the execution environment it provides, maintains these logical links. The main sources of

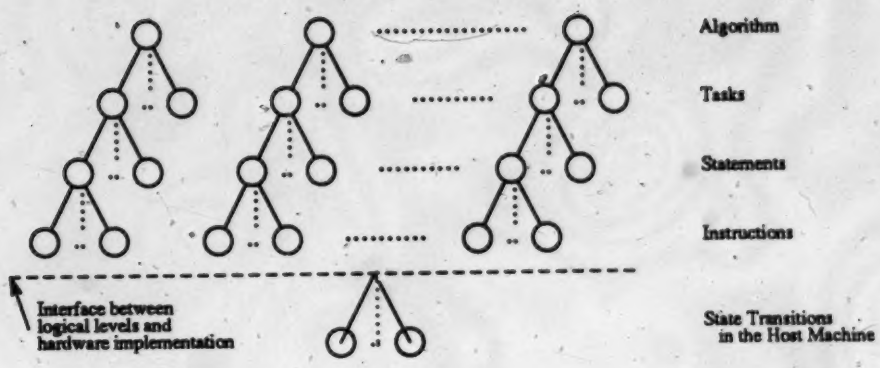


Figure 5.4 Structure of Problem Solution in Multi-programming Environment

complexity, which prevents the establishment of links between changes of state in the host machine and the triggering statements in the high-level language programs, are the heuristics used in scheduling and memory management, combined with the unpredictability of the behaviour of the competing programs in the execution environment.

Computing systems are capable of using many different combinations of resources and functional algorithms in performing any given job at any one time. The "precise" state of the machine is selected in "real" time with respect to the current distribution and availability of resources and the processing and resource demands being made by incoming jobs.

⇒ Computation consists of two logical classes of operations: control operations and memory manipulation. The former includes arithmetic and logical operations, the manipulation of local variables, and calls to peripheral processors or devices. The latter comprises the reading, manipulation, and writing of memory, specifically the program's data.

Fundamental functions of computers are determined by the two basic types of problems that account for most computer usage [Mat86, p. 275]. Computational-type problems (scientific, engineering, financial,

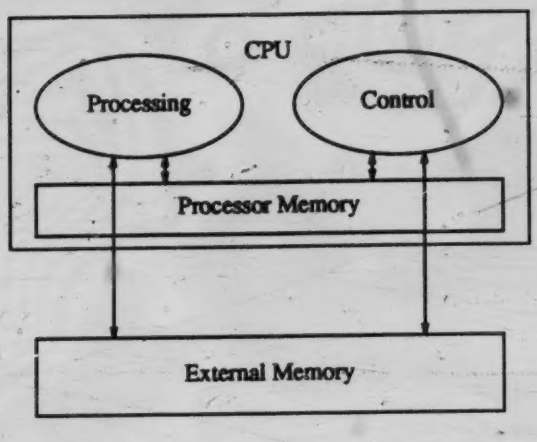


Figure 5.5 Program in execution view of memory

etc.) take some given parameters, perform arithmetic and logic processing on them, and produce a result. Data-based processing involves the retrieval of limited information from files, the alteration and/or processing of the information, and the possible storage of altered or new information in files. Other classes of problems tend towards one or the other of categories. The result is that the fundamental functions in all types of applications require accessing (reading and writing) information from some storage medium, and processing this information. We have two languages, or modes of operation. The outer mode is concerned with the flow of control, while the inner mode operates on the data.

We thus have two classes of operations performed in program execution (see Fig. 5.5). Control operations include arithmetic and logical operations, as well as I/O operations that are instigated by the CPU and eventually executed by peripheral devices or processors. These operations use strictly the resources of the processor and the data available in the processor memory. The other type of operation, by default, may be defined as memory accesses since processor operations are virtually cost-free. Whenever required data is not found in the processor memory, a specific memory access must be performed. This is a demand for resources external to the processor, and thus requires the intervention of external (to the processor) devices.

⇒ Programs in execution, particularly those written in highly structured languages, exhibit a high degree of locality in their address reference patterns.

That is, modules are small enough to fit on a few pages in the process' address space, and the local variables should be few enough to be assigned as register variables by the compiler (if these conditions are not met, it may be assumed that the function or module is doing too much work). If these conditions are met, the only references outside the immediate locality of the executing code will be to indexed variables.

⇒ Memory hierarchies have a noticeable, sometimes dramatic effect, on the performance of programs.

5.2.3.2. Relative Cost of Instructions

⇒ Instruction time is dominated by the time required to fetch an instruction and its operands from memory. As many as 10-20 internal operations comprise the interpretation of an instruction.

⇒ The cost of processor operations is determined by technology (the implementation of the processor) and data availability. Processor technology is constantly improving, and local data is available in registers (and cache) at close to processor cycle speeds.

An infinitely fast processor does not reduce the execution time for an algorithm to zero. Overall performance of the system is still limited by the number of memory accesses that must be made.

⇒ Operations on "external" data depend on the availability of that data, and often require intervention by an external device, the memory manager. External references require a minimum, physically bounded time to resolve.

Memory hierarchies were introduced to solve memory size and addressing problems in a more general way. One problem dealt with was the severe memory fragmentation that occurred in multiprogramming systems. (Fragmentation still occurs within the virtual address space because of fixed units of

transfer). The second problem confronted was the severe limitation of address space available to the user. In virtual memory, the user can be given an architecture-specified number of address bits, regardless of actual size of main memory. The hardware and/or software becomes responsible for moving information in and out of main memory when and if it is required, and for the logical equivalence of virtual and physical addresses.

The costs are significant. Hardware has to include special instructions for virtual memory management. There are instructions for loading addresses, for effecting the memory mappings, and for purging the cache and its directory. Address translation tables consume considerable main memory, limiting the size of the virtual address space that may be allocated to each user in a large multiprogrammed system. The entire main memory might be consumed by translation tables if all users were given the full virtual space available. Paging of page tables is a solution to this problem, but again at the cost of additional complexity. The cache gives the memory hierarchy the appearance of operating at close to the CPU cycle time. In a multi-processor configuration, when one processor's cache experiences a miss, the data may be in the cache of another processor. The cache coherency problem is also not trivial by any means. Controlling and managing the memory hierarchy requires substantial operating system resources, as well as hardware activity and assistance in the way of specialized architectures.

⇒ Processor cycle time is significantly smaller than memory cycle times. Memory cycle time thus dominates the cost of program execution.

Early efforts in hardware design focused on processors and storage. Since there was little difference between the speed of the two, developers were faced with improving the performance of both. As technology improved, however, logic speed increased greatly while memory speed *at reasonable cost* increased much less, relative to logic [Mat86, p. 207-208]. This resulted in a large gap, measured in orders of magnitude between logic delay and memory cycle time. In addition, main memory cycle time appears to be approaching a lower limit [Mat80].

Developments in memory technology and implementation have culminated in the use of memory hierarchies to provide virtually unlimited main memory. The cost appears in terms of a memory access time that is substantially greater than processor cycle time, and thus dominates the time required to execute programs. This memory access time appears to be reaching a lower limit, while processor technology is providing continuing improvement in processor cycle times.

5.3. Conclusion

Programs execute in complex multi-user and multi-programming environments. Because the model deals with execution cost and performance improvement, the computing environment for purposes of determining cost of program execution is defined in terms of the high-level language virtual machine. The high-level language specifies the "architecture" of the virtual machine, and represents the interface through which the user controls the operation of the machine, and thus may alter the structure of the computation and the cost thereof. The structure of the computer is visible to the user-as-programmer only through the operations specified by the high-level language. The dynamics, or underlying execution of operations specified in high-level languages resulting in state changes in the machine, are invisible to the user. There is no direct connection between operations specified by the user and changes of state in the machine because of the complexity of the implementation.

⇒ Program performance can be improved by choosing data structures and algorithms that minimize external data references.

As processor speed increases relative to memory speed, it becomes necessary to reorganize algorithms (solutions) so that the number of memory references is minimized.

Chapter 6

Application of the Model

6.1. Introduction

The generality of the model precludes the testing of specific hypotheses and the subsequent iterative refinement of the model. Instead, this chapter attempts to validate the base model assumptions. If the basic assumptions are correct, the model and the results will suggest directions for model refinement through further research. The model is not specific enough at this point to be very useful for answering questions raised by the results. Instead, the correctness of the basic assumptions means that the results must have an explanation in terms of the concepts of the base model.

A major component of the execution environment is the memory hierarchy provided. We first examine the effect on program performance of memory hierarchies. Evidence suggests that implementations of memory hierarchies are not transparent to executing programs. The memory hierarchy environment is simulated for the execution of four implementations of Minimum Spanning Tree algorithms..

In addition, the model makes claims about the relative costs of processor operations and memory references. Specifically, it is claimed that the cost of processor operations is negligible compared with the cost of memory accesses (as defined in Ch. 5). The cost of memory references is a function of both memory technology and the memory hierarchy implementation.

The model's contention regarding the direct relationship between memory references and performance is examined in two related case studies. Minimization of memory references is applied to a heuristic for the Traveling Salesman problem, first in combination with other improvements, then in isolation. The case studies include improvements at the design and implementation levels.

Finally, the improvements are analyzed to determine whether the model is able to predict the performance of the various implementations. A tentative refinement of the model is attempted by the inclusion of locality effects.

6.2. Effects of Memory Hierarchy on Performance

The cost model's assertion that virtual memory implementations are not transparent to executing programs is examined in a simulated environment. In this simulation, we adopt the notion of a dominant operation, in this case an access to lower (closer to the processor) levels of the memory hierarchy. We ignore accesses to the part of the program (code and data) that is independent of the size of the problem. An access to the lower memory level (read or write) has a fixed bandwidth and a set cost which is set equal to 1. Processor operations are assumed to have negligible cost and are ignored (See.LP [DoG88] for a more detailed discussion of this simulation).

In a two-level memory, not all the memory accesses to the lower memory levels are successful. In such cases, the reference must be resolved by accessing higher levels. The number of redirected accesses is a function of the sequence of references or locality of the instructions of the program, and the memory environment (size of levels in the hierarchy, unit of transfer, management policies). The cost of memory accesses may thus involve additional overhead, related to the number of redirected accesses.

The two experiments were run on a virtual memory simulator. One-level memory was simulated by allocating an infinite (read 'very large') memory to the running processes. As a result, no page faults were recorded, and all memory references were of equal cost. The memory hierarchy, or more precisely two-level memory, was simulated with a page size of 128 words, with 16 pages allocated to each running process.

The memory requirements of the four algorithms are not substantially different. However, when main memory is not sufficient for the needs of the program, the performance of two of the algorithms is affected drastically. The characteristics of the individual algorithms or their implementations are not as important here as is the effect of the change from 'real' one-level memory to 'virtual' one-level memory in the form of a memory hierarchy. Obviously designers and programmers concerned with program performance cannot afford to ignore these system implementation details.

6.3. Cost of Memory and Processor Operations

An additional factor in the dominance of memory references is considered to be the relative cost of operations. To test the assertion that the cost of memory references dominates execution time, a simple program was devised, comprising a loop with ten identical statements, operating on different data types. Two operations, assignment and addition, are each performed on three different data types: simple variables and elements in one- and two-dimensional arrays. The six programs were executed in two different computing environments, a VAX-11/780 and a MIPS M/1000 Development System.

No attempt was made to perform the measurements in the context of a "typical" computing job (if such exists), or to measure the full range of operations (arithmetic, logical, I/O, etc.). This may limit somewhat the usefulness of the results; however, the test programs do represent minimal programs and thus incur all the overheads associated with program execution.

The results confirm the basic assertion of the cost model. The theoretical complexity of the three program fragments involving addition are identical. The differences in execution times are related to the type of variable being operated on, rather than the type of operation being performed. The constant associated with the addition operation varies from 1 to approximately 30 on one system, and from 1 to 10 on the other system. The comparison between high and low values, and between addition and assignment operations, suggests that the cost of the operations themselves is minimal, and that the intrinsic complexity of different operations is not reflected in program performance. The difference between fragments 2 and 6, for instance, is not in the operation, but in the type of operands on which the operation is performed. In terms of the model, the difference is that fragment 5 references *external* memory, whereas fragment 2 operates on *internal* memory.

The range of constants is expected to be different because of the characteristics of the respective systems. What is puzzling, however, is the range of the constant for operations on elements of two-dimensional arrays, and the similarity (in all but absolute value) of the curves. The anomalies and their

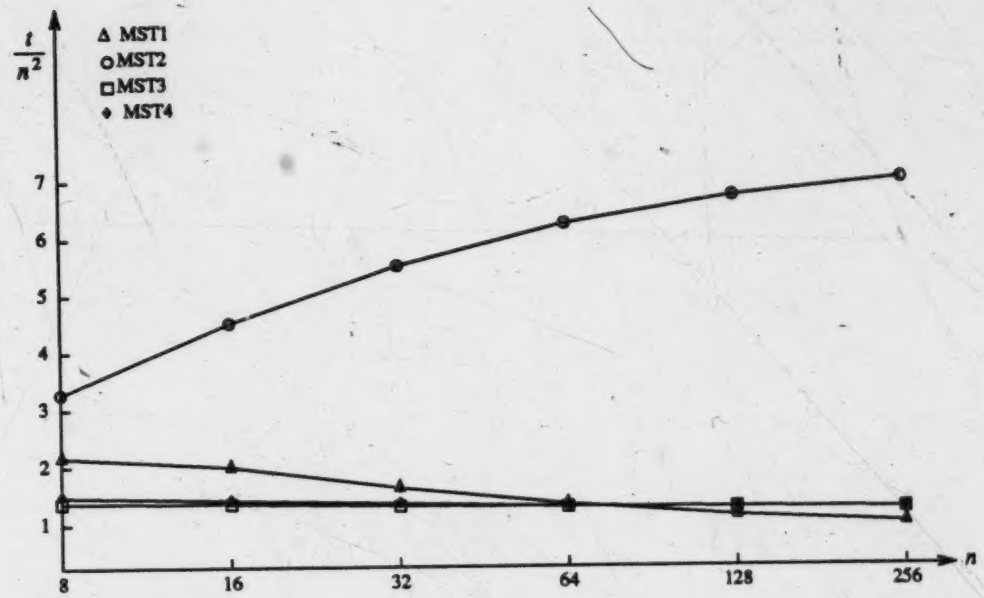


Figure 6.1 Performance in (Simulated) One-level Memory

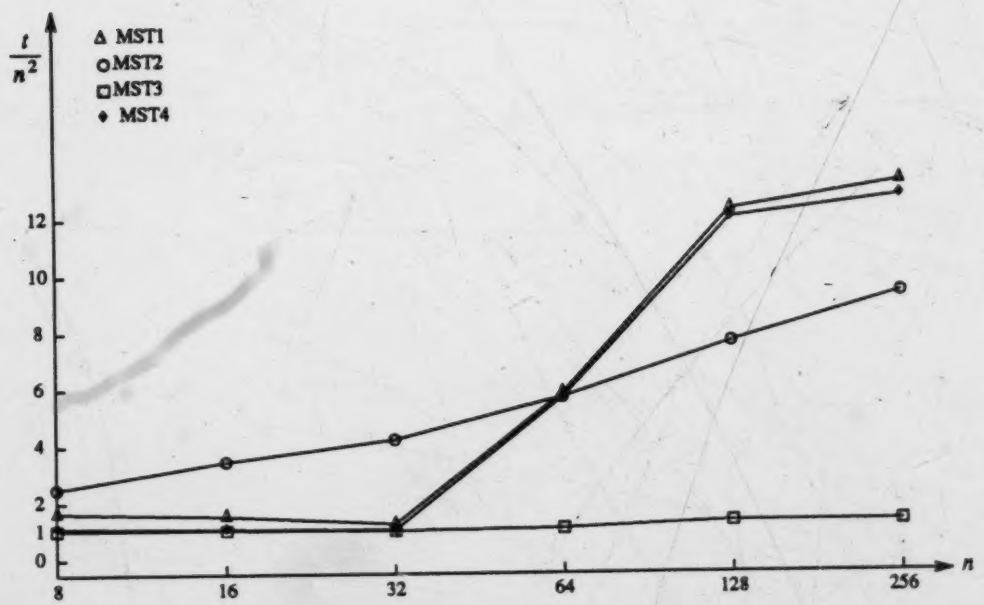


Figure 6.2 Performance in (Simulated) Memory Hierarchy

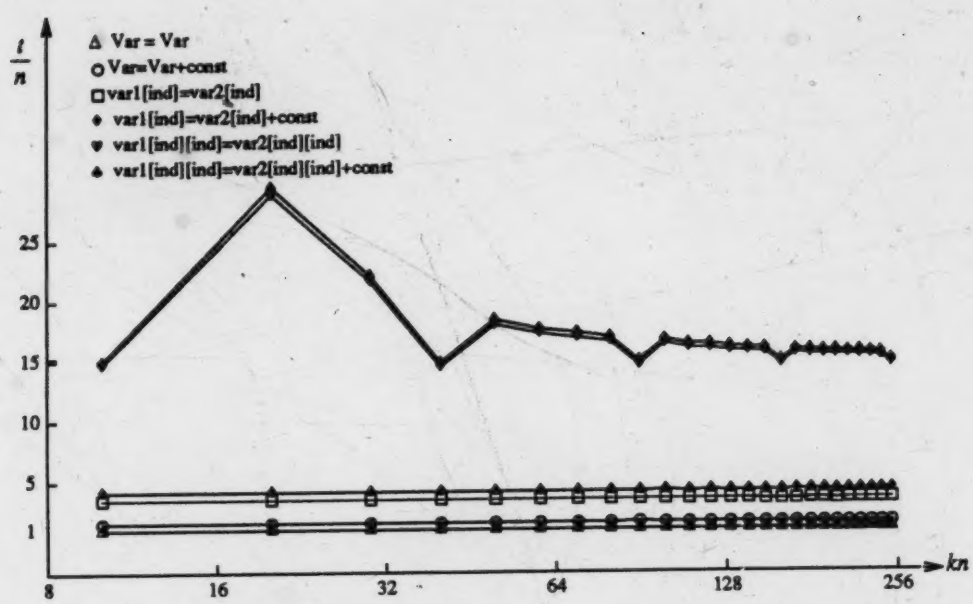


Figure 6.3 Relative Cost of Operations (VAX-11/780)

relative magnitudes suggest two divergent possibilities. Firstly, based on the other results, the anomalies must be attributed to the referencing of the array elements, rather than to the operations being performed. Secondly, it appears that there is a basis for assuming that common techniques for accessing memory do exist among different machine implementations, and that these implementations behave in a similar fashion for programs of a specific size. It is not clear how this is achieved considering the differences in architecture and configuration, even given the similarity in operating systems.

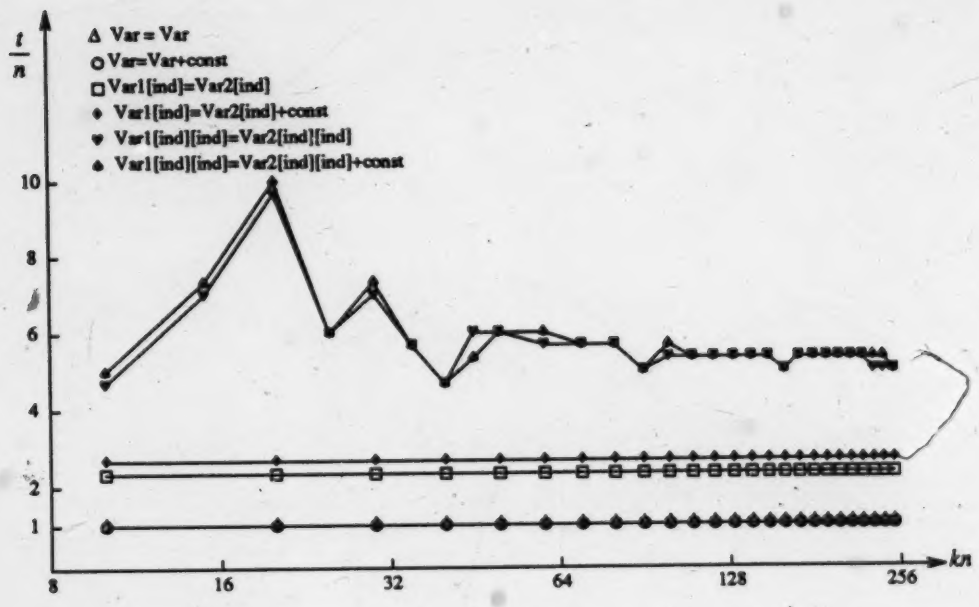


Figure 6.4 Relative Cost of Operations (MIPS)

6.4. The Model Applied

The problem of efficiently finding a tour of a set of points with absolutely minimal total length is the subject of much research and activity. We are concerned in this case only with quickly finding a relatively good, if not optimal tour. The **Nearest Neighbour** heuristic for the Traveling Salesman problem, starting from an arbitrarily chosen starting point, repeatedly visits the unvisited point nearest to the current point until all points have been included in the tour. The algorithm has been observed to give tours whose lengths are usually within about 25% of optimal.

These case studies were executed in two different environments. The one-dimensional version according to Bentley (case study 1) was executed on a VAX-11/780. The two-dimensional implementations (case study 2) were executed on the MIPS M/1000 Development System. The statistics presented are in terms of user time only.

6.4.1. Case Study 1 - Integration with Other Improvements

The first case study follows the sequence of improvements proposed and discussed by Bentley. Bentley's implementation is in a Pascal dialect, executed on a PDP-11; therefore comparisons with his results were considered inappropriate at this point. The mixture of minimization of external memory references and other improvements obscures somewhat the thrust of the model's application; however, it does provide a perspective on the relative value of the model's application. Because the model is aimed at the *practitioner*, it is important to relate the model to the general practice of computing.

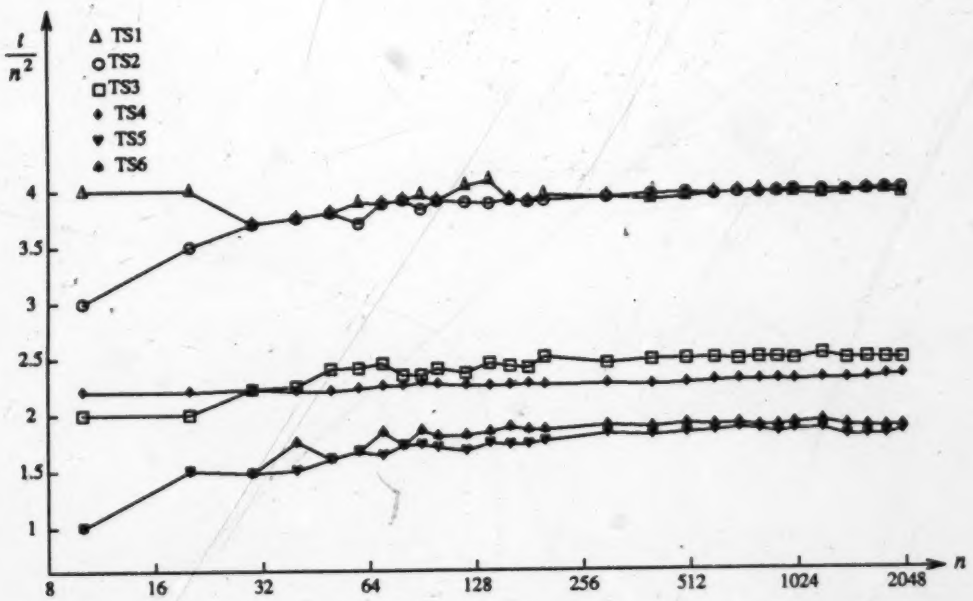


Figure 6.5 Case Study 1: Traveling Salesman (VAX-11/780)

The first implementation (TS1) represents the basic algorithm. For an arbitrary point, the nearest neighbour is found by calculating the distance to every other point not so far included in the tour. The code is a direct implementation of the algorithm with no concessions to performance optimization. The algorithm is of order n^2, and the implementations diverge only marginally.

```
for( I = 1; I < NumPts; I++ )
{
    CloseDist = MAXINT;
    for( J = 1; J < NumPts; J++ )
    {
        if ( Visited[J] != TRUE )
        {
>           if( Dist( ThisPt,J ) < CloseDist )
>           {
>               CloseDist = Dist( ThisPt,J );
                ClosePt = J;
            }
        }
    }
    Visited[ClosePt] = TRUE;
    ThisPt = ClosePt;
}

function Dist( I,J )
int I,J;
{
    return( sqrt( sqr (PtArr[I].X - PtArr[J].X )
                + sqr (PtArr[I].Y - PtArr[J].Y )));
}
```

The second version (TS2) aims to improve performance by eliminating in each iteration of the inner loop a reference to "external" variables (related to the size of the problem), and replacing it with a reference to a variable independent of the size of the problem. The lines marked with '>' are replaced with the following:

```
ThisDist = Dist( ThisPt,J );
if( ThisDist < CloseDist )
{
    CloseDist = ThisDist;
```

The result of the "improvement" is marginal, if not non-existent. Superficially, it may seem that the elimination of the second call to the function **Dist** and the resulting elimination of the second access to the coordinate records in the function will result in a substantial improvement in performance. A closer look at the two implementations reveals two factors that will minimize the effect of the change. The inner loop is executed only if the new distance is smaller than the previous smallest distance. The average number of

times this section of code is executed for M points is $H_M = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{M}$, called the M-th harmonic number, approximately equal to the natural logarithm of M [Ben82, p. 16,136-139]. H_{1000} equals about 7.5.

A second factor is locality of both time and space. A second reference to the same location should find the referenced data for one of two reasons. It is highly likely that most of the code and data for a small program such as this one will be resident most, if not all the time. Even if this assumption should prove to be false, it is highly probable that the page containing the data will still be resident as a result of the previous reference.

The third version (TS3) makes an improvement unrelated to the cost model. The function **Dist** is simplified by eliminating a call to the system function to calculate the square root. This simplification is based on the observation that any ordering between two non-negative quantities is maintained between the squares of those quantities. The new function is named **DistSqrd.**

The improvement in performance is more substantial, the difference being almost a factor of two. The calls to the **sqrt** function are removed altogether, providing clear evidence of the overhead involved in such system calls.

Version four (TS4) is based on the observation that there is no need to look at all points to determine the nearest when we have a limited number of unvisited points. This change is more substantial, and might be classified as a selection of data structures, and therefore at a higher design level. The initialization is somewhat different, but the overall structure of the routine is almost unchanged. When the next closest point has been identified it is swapped with the point in **UnVis[HighPt]** and **HighPt** is decreased by one. This maintains the invariant condition that all unvisited cities can be found in **UnVis[1..HighPt].**

```
for( I = 0;I < NumPts ;I++ )
    UnVis[I] = I;

....
```

```
while( HighPt > 0 )
{
    CloseDist = MAXINT;
    for( I = 1; I < HighPt; I++ )
    {
        ThisDist = DistSqrd( UnVis[I],ThisPt );
        if ( ThisDist < CloseDist )
        {
            ClosePt = I;
            CloseDist = ThisDist;
        }
    }
    ThisPt = UnVis[ClosePt];
    SwapUnVis( UnVis,ClosePt,HighPt );
    HighPt = HighPt - 1;
}
```

The change in the design of the algorithm in TS4 leads to a small reduction in execution time (approximately 6%). It may have been expected that the change would show more dramatic results. The additional processing required to maintain the array **UnVis** offsets a substantial portion of the savings effected by reducing redundant execution in the two loops.

The next version (TS5) re-writes the procedure call to *DistSqrd* in-line, and assigns invariants outside the inner loop instead of re-evaluating them each time through the loop.

```
while( HighPt > 0 )
{
    ThisX = PtArr[ThisPt].X;
    ThisY = PtArr[ThisPt].Y;
    CloseDist = MAXINT;
    for( I = 1; I < HighPt; I++ )
    {
>       ThisDist = pow(( PtArr[UnVis[I]].X - ThisX ),2 )
>           + pow(( PtArr[UnVis[I]].Y - ThisY ),2 );
>       if ( ThisDist < CloseDist )
>       {
>       ClosePt = I;
>       CloseDist = ThisDist;
        }
    }
    ThisPt = UnVis[ClosePt];
    SwapUnVis( UnVis,ClosePt,HighPt );
    HighPt = HighPt - 1;
}
```

Another noticeable improvement (approximately 19%) is effected by the re-writing of the DistSqrd procedure in-line and the removal of invariants from the inner loop (TS5). The re-arrangement eliminates the overhead of a procedure call, but leaves intact the processing performed in the procedure. The major improvement is likely the elimination of the redundant references to the coordinate record of the current point.

The last version (TS6) postpones the calculation of the total distance between two points until it is determined that the x-distance is not by itself too large to keep the point in consideration. The lines marked '>' are replaced with:

```
ThisDist = pow(( PtArr[UnVis[I]].X - ThisX ),2 );
if( ThisDist < CloseDist )
{
    ThisDist = ThisDist + pow(( PtArr[UnVis[I]].Y - ThisY ), 2 );
    if ( ThisDist < CloseDist )
    {
        ClosePt = I;
        CloseDist = ThisDist;
    }
```

The last modification in this sequence (TS6) separates the references to the coordinates of the point under consideration for inclusion in the tour. Consideration of the x-distance alone may eliminate the point from consideration. Heuristic analysis and experimental evidence [Ben82] suggest that the number of times the y-distance must be considered is only about $2.25M^{\frac{1}{2}}$, (which is approximately 70 for M=1000). The results suggest that this change is counter-productive. This is perhaps a function of locality - the postponed access is to a location that has just been referenced - and the additional conditional statement required.

Results suggest tentatively that minimization of memory references results in improved performance. An effect not accounted for by the model, but used to explain some of the results, is locality. The effects of locality are not always well-understood, but in a relatively small and simple program such as this its effects may be more easily seen. Changes at both the design and implementation level were tested and found to be successful in improving performance.

6.5. Case Study 2 - Memory Reference Minimization

A modification of Bentley's program is implemented to test the model's application independent of other improvements. Instead of a one-dimensional array storing the coordinates of each point, this version stores the distance between points I and J in the location in a two-dimensional array indexed by I and J. The algorithm is the same, but the code is simplified because the distances do not have to be calculated. These implementations were executed on the MIPS M/1000 Development System. The algorithm again is order n^2, but this time the results do not conform to this prediction of performance

Version 1 (TS1x) is a direct translation of the algorithm without considering efficiency.

```
    for( I = 1; I < NumPts; I++ )
    {
        CloseDist = MAXINT;
        for( J = 1; J < NumPts; J++ )
        {
            if ( Visited[J] != TRUE )
            {
>                   if( Dist[ThisPt][J] < CloseDist )
>                   {
>                       CloseDist = Dist[ThisPt][J];
                        ClosePt = J;
                    }
            }
        }
        Visited[ClosePt] = TRUE;
        ThisPt = ClosePt;
    }
```

The second version (TS2x) aims to improve performance by eliminating in each iteration of the inner loop a reference to "external" variables (related to the size of the problem), and replacing it with a reference to a variable independent of the size of the problem. Reference to the same location in consecutive statements is avoided by assignment to a local variable. The lines marked with '>' are replaced with the following:

```
ThisDist = Dist[ThisPt][J];
if( ThisDist < CloseDist )
{
```

```
CloseDist = ThisDist;
```

The last version (TS4x) of this implementation reduces accesses by maintaining a list of unvisited points, as in the fourth version of the first case study.

```
while( HighPt > 0 )
{
    CloseDist = MAXINT;
    for( I = 0; I < HighPt; I++ )
    {
        ThisDist = Dist[UnVis[I]][ThisPt];
        if ( ThisDist < CloseDist )
        {
            ClosePt = I;
            CloseDist = ThisDist;
        }
    }
    ThisPt = UnVis[ClosePt];
    SwapUnVis( ClosePt,HighPt );
    HighPt = HighPt - 1;
}
```

The results show more clearly the effects on performance of the minimization of memory accesses. This is predicted by the results of the comparison of the costs of basic operations (Figs. 6.1-2). Again, the improvement from TS1x to TS2x is slight or non-existent. As before, the improvement is limited in effect because of the number of times that the inner statements are executed. The results of the second improvement are more substantial.

An interesting phenomenon here is the lack of distinction in performance among the three versions for values of N greater than 1000, and an apparent leveling of the curves at that point. The first is assumed to be the result of having two indices to take into account in the calculation of the address. Adjacent elements in the array may not be physically adjacent, or on the same page. This effect should be more noticeable as the number of dimensions increases. This is perhaps because the reference pattern becomes relatively more localized, especially if large programs are allocated more main memory and cache.

Curve fitting shows that the three curves, but particularly that for TS3x, conforms to approximately $n^2 \log_{10}(\frac{n}{1.6})$. It is interesting to note that the complexity as predicted by the logarithmic cost model is

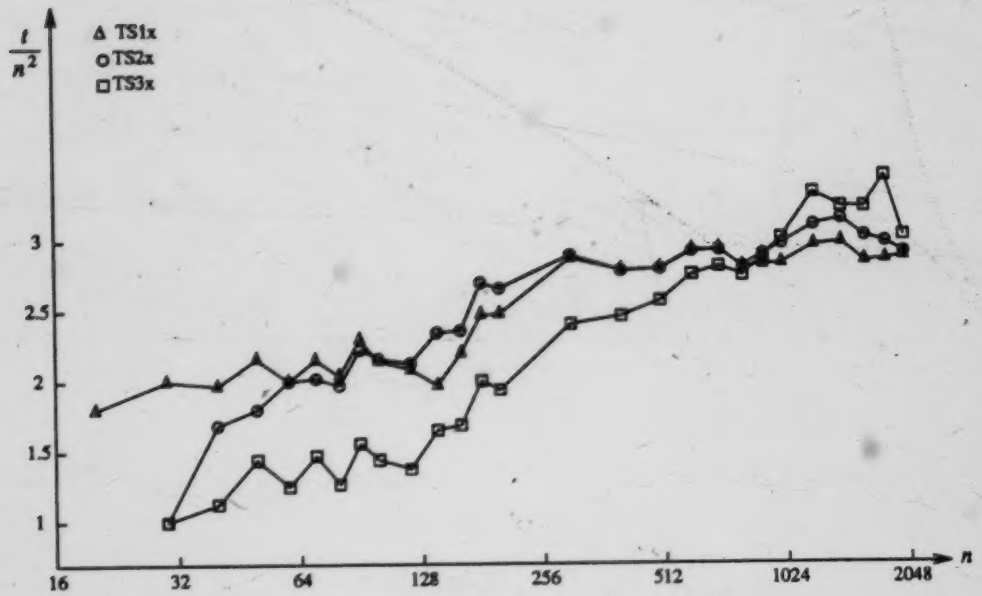


Figure 6.6 Case Study 2: Traveling Salesman (MIPS M/1000)

$n^2 \log n$, instead of the n^2 predicted by the uniform cost model. Other evidence suggests, however, that this is coincidental, and that further speculation on this aspect of the results is unwarranted.

6.6. Performance Analysis and Prediction

If the model is to be *effective*, it must provide the means for analyzing and predicting the performance of a program and for evaluating proposed improvements. The model assumes no difference between reading from and writing to external memory. All memory references are considered to have equal cost in the model. We can analyze the test programs to determine how they will perform, if not in absolute terms (user time), then relative to the other programs in the test suite. The functions describing the performance of the programs include the constants, as these are considered to represent implementation details which should not be ignored. (Because the improvement in TS3 does not involve memory references, the predicted performance is based on the experimantal results) Figure 6.7 shows the results for case study 1.

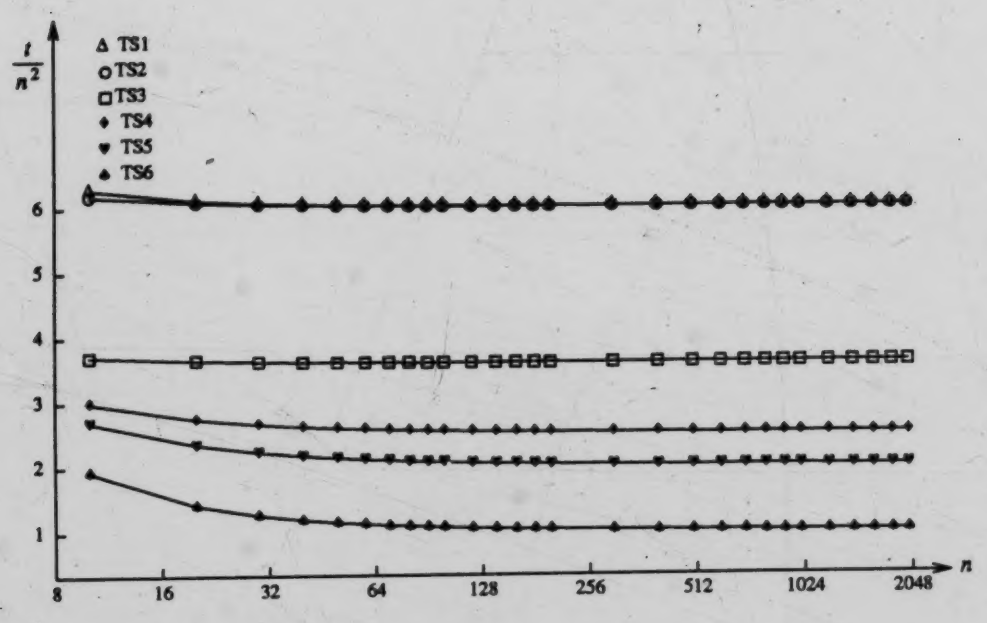


Figure 6.7 Cost Model's Performance Prediction (Case Study 1)

The relative magnitude and range of execution times is predicted correctly (compare with Fig. 6.5). The minimal difference in performance between TS1 and TS2, including the initial divergence, is predicted, as are the improvements from TS2 to TS4 and TS4 to TS5. The predicted improvement in TS6 is not borne out in the experimental results. As in TS2, locality is the likely reason. The actual improvement is valid - in only H_n cases will the y-component need to be calculated.

Because locality is a known factor in the functioning of memory hierarchies, it was deemed appropriate to include an ad hoc locality effect in the model's performance prediction. In the relatively simple case of case study 1, repeated references to the same location are relatively easy to identify. If we include in the model some simple assumptions about locality, the predicted performance is much closer to the experimental results. Combining adjacent references to the same address produces the following predictions of performance. The magnitude of the individual improvements from TS1 to TS6 is clearly reflected when locality effects are taken into account, as is the size of the overall improvement from TS1 to TS6. Particularly

noticeable is the change in the predicted result from TS5 to TS6. The analysis of the experimental results suggests that locality was likely a factor in the relative performance of these two implementations, because the only improvement was the elimination of a reference to a location that had just been accessed.

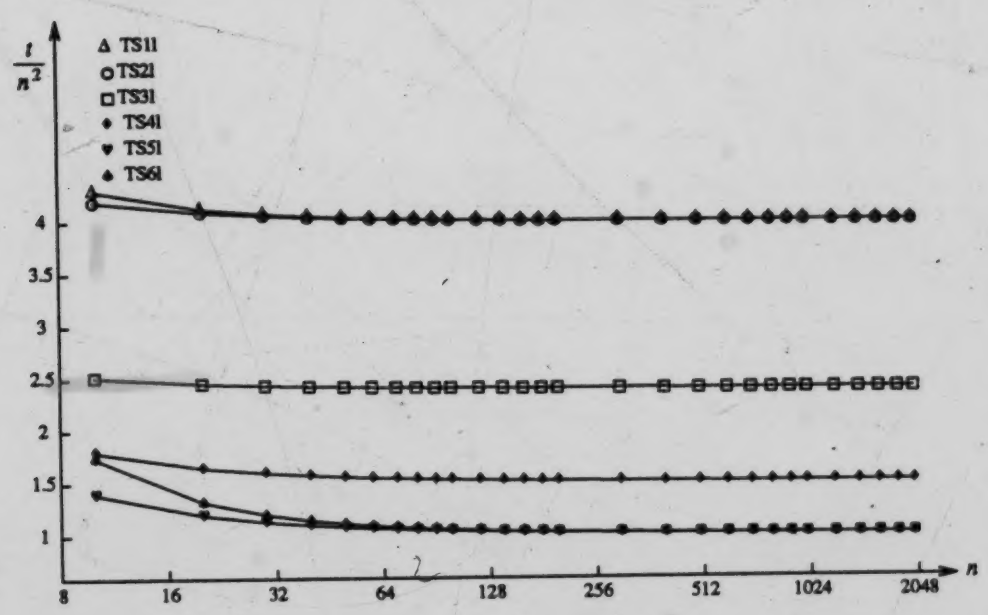


Figure 6.8 Cost Model's Performance Prediction (Including Locality)

6.7. Conclusions

As suggested by evidence presented in Chapters 2 and 4, the implementation of memory hierarchies is not transparent to programs in execution. Programmers may not have to write special routines to manage memory for their programs, but they should definitely be aware of the effects of memory reference patterns on performance. This is definitely an advance over the problems of writing overlays, but it brings into question the appropriateness of the term 'virtual' memory if it is intended to suggest that memory hierarchies act like one-level memories.

Results confirm expectations about the relative cost of processor operations and memory references. Questions remain about the source of the disparity between accessing elements of arrays with different numbers of dimensions. There is no doubt, however, that the software and hardware costs of memory hierarchy implementations and the relative improvements in processor and memory technologies combine to make memory accesses the dominant operation in the execution of programs.

Minimization of memory references tends to improve the performance of programs. Changes at both the implementation and design level were tested to determine the validity of this claim. In both case studies, improvements suggested by the model resulted in improvements in performance. Some of the suggestions, when examined in the light of the concept of locality, proved to be less productive than others. Although the model is too general to deal with the effects of locality, it is an obvious factor in any study of program behaviour in memory hierarchies.

We exclude optimizing compilers in the consideration of increased efficiency of programs for two reasons. Firstly, the effects of an optimizing compiler cannot be predicted by the programmer. This includes effects such as the address calculations for data structures whose size is a power of 2. Secondly, the programmer cannot affect its operation. No set of rules exists that, when followed, will lead to a *predictable* optimization of the executable code. This excludes optimizing compilers from consideration in this performance model.

The model proves to be capable of predicting performance for at least simple programs. The magnitude and quality of improvements in one case study were successfully predicted. When an ad hoc locality effect was incorporated in the performance prediction, the predictions were even closer to the experimental results.

Questions remain because of the limited scale of the testing, and because some of the results raised more questions than they answered. There is no doubt, however, that a cost model of program execution should consider memory references to be the dominant operation, that the minimization of memory references leads to improved performance, and that to some extent the performance of programs may be

predicted.

Chapter 7

Evaluation and Conclusions

7.1. Introduction

This thesis was undertaken to investigate the possibility of predicting the performance of high-level language programs. Software engineering neglects this area, and complexity analysis is aimed at abstractions rather than at the practitioner. The approach chosen was the development of a descriptive base model to allow the testing of general propositions about the environments in which such programs execute. This base model is derived from knowledge about characteristics of components and subsystems of computing systems, and by reference to experimental results describing their behaviour. The model is constructed according to design methodologies for complex systems, from the perspective of the user-as-programmer.

The underlying concern of the base model is the complexity of the computing environment. It is possible to study (through traces or simulation) the behaviour of the cache, of the rest of the memory hierarchy, and of the scheduler, but the interaction of these subsystems is likely to produce behaviour that will prove intractable under reductionist analysis. The environment is not isolated from the processes it serves - it is partly defined by the interaction of its various processes with the processes for which it provides the required resources. An alternative approach is to study the environment in terms of the resources that it manages and provides on demand. To study the performance of programs, these resources are defined by the operations and data structures specified by the programmer in high-level language programs. Execution of a program in the environment may be seen as the simulation of a machine with capabilities and resources specified in the program's algorithm and data structures.

The thesis investigates performance to construct a model to aid in designing algorithms for efficiency of performance. This requires a concentration on the important aspects of the problem, which may be defined as the specification of the algorithm in terms of the operations and data structures required to represent and solve the problem, and in terms of the costs involved in computing. These problems are

interdependent. The representation of a solution in terms of the model should enable the analyst/designer to determine clearly the relative cost of a particular solution.

We examine the particular problems associated with determining the cost of *implementing* a particular solution. It is aimed at practitioners of computing. The thesis proposes that the cost of computing is dominated by references to memory locations outside the immediate locality of the program's code and data. It does this by proposing that program performance may be improved by minimizing references to this "external" memory. This thesis is tested by determining the relative costs of operations and memory references, simulating program execution in an environment in which the cost of memory references dominates execution time, and applying the basic proposition to program improvement in a real computing environment. This chapter examines the results of these tests with the goal of determining the extent of the validation of the model. Based on the results, and the tests performed, directions for further research are examined. Lastly, the importance of this problem and the results to the discipline of computing science are discussed.

7.2. Model Validity

Descriptive models are by definition not designed to generate results about the operational detail of the system being modeled, but about the general characteristics of that system. In a model concerned with performance, the identification of a dominant operation is of primary concern. Based on hardware characteristics and software implementation, the model asserts that performance of a program is a function of references to "external" memory (elements in indexed data structures).

Experiments in two different environments show that execution times of simple operations is a function of the type of operand. Operations on simple variables are virtually cost-free compared to operations on indexed data elements. These results illustrate tendency rather than demonstrating a fixed relationship, although for simple programs a definite relationship appears to exist. The model does not explain the noticeable difference in cost between accessing elements in one- and two-dimensional structures, nor does

it deal with the performance anomalies in the latter case. In addition, the results are obviously difficult to evaluate in isolation from system characteristics and implementation. The tendency is enough, however, to lend creedence to the assertion that the memory reference is the dominant operation in program execution.

Case study 1 tests the active assertion that program performance may be improved by reducing external memory references. The results are ambiguous. Analysis predicts that some of the alterations will improve performance marginally, if at all. These predictions are not borne out in all cases. The common factor is the suggestion that locality of time and place plays a large part in determining the cost of a memory reference. References to memory locations recently accessed, or in proximity to these, will be virtually cost-free. Eliminating these references appears to be non-productive. This is confirmed when a minimal notion of locality based on proximity of references is introduced into the model. Since little is known about the locality properties of most algorithms, the importance of these results should not be overestimated. Most algorithms do not have reference patterns as obvious as in these simple test programs. Nevertheless, the effects cannot be denied.

Case Study 2 tests the same assertion in simpler implementations of the same algorithm. No calculation of distances is required, but the cost is a more complex data structure (two dimensions). Improvements in performance are more pronounced, as suggested by earlier results on the comparative costs of operations. The results are again problematic. The differences in performance between implementations are no longer apparent for values of N greater than 1000 (the size of the executable makes it difficult to pursue this - i.e. the problem size is unlikely to exceed 2000 using this algorithm). It is not clear whether this trend continues for values of N greater than 2000. In addition, the relationship between performance and problem size is no longer linear as in the first case study. Accessing elements in a two-dimensional array appears to be subject to an additional complexity factor. It is again not clear whether this trend continues for larger values of N.

The model may not be able to deal with the differences between various implementations of virtual one-level memory in memory hierarchies. Different implementations (relative sizes of levels, algorithms,

etc.) have specific performance goals. They are based on modes of program behaviour or system user requirements, and will thus obviously affect the performance of programs in such environments. It remains to be seen whether a general model can account for these differences. The differences between the shapes of the curves of theoretical and experimental results points to the effect of implementation details that are obviously not reflected in the model. Various anomalies in the experimental results need to be examined to see whether they can be described systematically.

7.3. Application of the Model

The model is general, because it deals with logical resources, therefore should be applicable to all von Neumann model computing. Programs (and algorithms) can be analyzed by replacing indexed memory references by some appropriate function call (get or put). The rule can be applied in (a) system design (module level) to structure a system so that memory references are minimized, (b) in algorithm design to choose structures and algorithms, (c) in program implementation to choose among the various ways of implementing the algorithm and data structures, and (d) in program improvement to improve the performance of critical areas (although other techniques are applicable here as well). The inclusion of the effects of locality is not done systematically, because this has not been investigated sufficiently and the effects are not well-understood in the context of this model.

Other techniques are not excluded. The value of good programming skills is not denied by this model. The model is general. This is its primary advantage over long lists of individual techniques. It is also machine- and configuration independent. Machine and configuration dependent approaches should be left to mechanical techniques implemented for specific systems. Note problems with transportability, compatibility, which should make generality attractive even at the cost of marginally lower performance.

Maximum performance from an algorithm can only be obtained when it has been coded with a particular machine in mind. The rules for efficient programming as proclaimed by Bentley [Ben82, p.145-151] require 7 pages to summarize. This is an indication of the complexity of the problem, particularly since not

all rules apply in all situations and/or environments. The application of this type of approach sacrifices the clarity of the original algorithm.

7.4. Further Research Directions

By design, the descriptive model raises more questions than it answers. Two essential aspects of the model are confirmed, albeit not conclusively, by the experiments conducted. The high-level language interface provides a satisfactory abstraction for the analysis and prediction of the performance of programs in complex environments. The extent to which this level of abstraction will be adequate is not clear. The examples used were simple, and the results not always conclusive. There are, however, grounds for proceeding both with more difficult examples and other languages to pursue the question of the generality of the model.

On an immediate level, some questions relevant to the experiments conducted should be answered to provide a more comprehensive understanding of the nature of programs and environments. The comparison of the costs of various operations might be extended to other arithmetic and logical operations. A particular question to be resolved concerns the rising (as N increases) cost of memory accesses in two-dimensional structures. It is suspected that additional indices act as additional levels of indirection - that the address now depends on another index, and thus has a greater chance of residing outside the current resident set.

Further experimentation will perhaps lead to more insight into the mysteries of the environment. Specifically, the difficult areas involve the interaction between the memory management policies in force at various levels of the memory hierarchy, job scheduling policies, and the size and nature of the workload. If, as is claimed by some, the complexity of programs reflects the complexity of the execution environment, the interaction between these components must be investigated.

Memory hierarchies attempt to give some appearance of an ideal system at low cost, but with severe performance limitations (compared with the ideal), and great complexity. Memory hierarchies are not the

same as the virtual one-level memories that they attempt to implement. We need to look at what factors in their design, implementation, and operation account for the differences, and whether it is possible to generate general rules to describe the behaviour of this "virtual" memory. It is suspected that dominant factors will be the relative sizes and organization of the hierarchy components, and the level of multiprogramming. Current computing system design is greatly influenced by the concept of virtual memory. The shape of existing systems perhaps has too great an influence on the design of future systems. It might be interesting to speculate on the form of virtual memory if it were designed only with reference to available technology and requirements, excluding historical designs and implementations.

Also look at further use of system science techniques to facilitate analysis of other complex information technologies or applications. The approach has been applied productively to a wide variety of problems and situations. Systems science techniques are particularly suited to the study of the dynamic behaviour of systems and system-wide effects. For instance, optimizing cache performance may cause degradation in other areas of the memory hierarchy because of the use of common subsystems. No study of an isolated subsystem will adequately account for this.

7.5. Conclusion

The proposed model is based on the recognition of a distinction between computation and computing, between theory and practice. Computation is a mathematical action, emphasizing operations. This is the approach taken by complexity analysis, determining difficulty. Computing, on the other hand, is an engineering discipline. As such, it is concerned with the process required to perform a computation mechanically. This process is dominated by the memory transactions ignored in formal descriptions of computations. The highly abstract quality of the computational model on which the design of computers was based made it easy to introduce mathematics into the study of their theory. The distinction between theory and practice, computation and computing, was easily obscured, resulting in serious misapprehensions about the practice, as opposed to the theory, of computing. The application of algorithmics is not lim-

ited to computing science; we must therefore be sure that the details of the application reflect the realities of the application. No purpose is served by sacrificing applicability to tractability.

Robert Tarjan suggests that the gap between theory and practice be bridged by an increased emphasis on the experimental analysis of algorithms. Any activity in this area must take into account that operating systems provide environments based on assumptions about program behaviour, and that program behaviour is analyzed in terms of some computing environment. Rather than focus on only the components of a complex system, a predictive model should be more than a simple summation of the effects of the components, and should account as well for the interactions effects. (therefore system level is correct approach). May even be applicable to non-von Neumann architectures because deals with logical resources.

Knowledge of abstract computational practice provides a framework for thinking about the practice of computing, but computing science should offer guidance for practitioners as well, rather than just depending on spill-over from abstract knowledge about computing.

References

[Ack64] Russell L. Ackoff, General System Theory and System Research: Contrasting Conceptions of Systems Science, in *Views on General System Theory*, M. D. Mesarovic (ed.), John Wiley, 1964, 51-60.

[AHU74] Alfred V. Aho, John E. Hopcroft and Jeffrey D. Ullman, in *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[AEH84] Timo O. Alanko, Hannu H. A. Erkio and Ilkka J. Haikala, Virtual Memory Behaviour of Some Sorting Algorithms, *IEEE Transactions on Software Engineering SE-10*, 4 (July 1984), 422-431.

[Arb72] Michael A. Arbib, *The Metaphorical Brain: An Introduction to Cybernetics as Artificial Intelligence and Brain Theory*, Wiley Interscience, 1972.

[BeN71] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, 1971.

[Ben82] J. L. Bentley, *Writing Efficient Programs*, Prentice-Hall, 1982.

[BoB85] B. A. Bowen and W. R. Brown, *Systems Design*, Prentice_Hall, 1985.

[BrG68] Barbara S. Brawn and Frances G. Gustavson, Program Behaviour in a Paging Environment, *Proc. AFIPS 1968 Fall Joint Comput. Conf. 33*, (1968), 1019-1032.

[Cam82] Jeremy Campbell, *Grammatical Man: Information, Entropy, Language, and Life*, Simon and Schuster, 1982.

[Com67] L. Comeau, A study of the effect of user program optimization in a paging system, *ACM Symposium on Operating System Principles*, Oct. 1-4, 1967.

[Das82] Subrata Dasgupta, Computer Design and Description Languages, in *Advances in Computers*, vol. 21, Academic Press, 1982, 91-150.

[Den70] Peter J. Denning, Virtual Memory, *Computing Surveys 2*, 3 (September 1970), 153-189.

[DoG88] Wlodzimierz Dobosiewicz and Pawel Gburzynski, Complexity of Algorithms in an Hierarchical Memory Environment, In preparation, 1988.

[Dor79] R. W. Doran, *Computer Architecture: A Structured Approach*, Academic Press, 1979.

[Fly77] Michael J. Flynn, Computer Organization, in *High Speed Computer and Algorithm Organization*, David J. Kuck, Duncan H. Lawrie and Ahmed H. Sameh (ed.), Academic Press, 1977, 41-70.

[Gen73] W. M. Gentleman, On the Relevance of Various Cost Models of Complexity, in *Complexity of Sequential and Parallel Numerical Algorithms*, J. F. Traub (ed.), Academic Press, 1973, 103-110.

[Goo83] James R. Goodman, Using Cache Memory to Reduce Processor Traffic, in *The 10th Annual Symposium on Computer Architecture*, Computer Society Press, 1983, 124-131.

[Gra75] R. M. Graham, Performance Prediction, in *Software Engineering*, G. Goos and J. Hartmanis (ed.), Springer Verlag, 1975, 395-462. Lecture Notes in Computer Science (30).

[Har87] David Harel, *Algorithmics : The Spirit of Computing*, Addison-Wesley, 1987.

[HaG71] D. J. Hatfield and J. Gerald, Program Restructuring for Virtual Memory, *IBM Systems Journal 10*, 3 (1971), 168-192.

[HoR77] R. W. Hon and D. R. Reddy, The Effect of Computer Architecture on Algorithm Decomposition and Performance, in *High Speed Computer and Algorithm Organization*, David J. Kuck, Duncan H. Lawrie and Ahmed H. Sameh (ed.), Academic Press, 1977, 411-.

[HoS78] Ellis Horowitz and Sartaj Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, 1978.

[KaW73] K. R. Kaplan and R. O. Winder, Cache-based Computer Systems, *IEEE Computer 6*, 3 (Mar. 1973), 30-36.

[Kli69] George J. Klir, *An Approach to General Systems Theory*, Van Nostrand Reinhold, 1969.

[Kor66] R. R. Korfhage, *Logic and Algorithms*, John Wiley & Sons, 1966.

[KuR68] C. J. Kuehner and B. Randell, Demand paging in perspective, *Proc. AFIPS 1968 Fall Joint Comput. Conf. 33*, (1968), 1011-1018.

[Lau82] Edwin J. Lau, *Performance Improvement of Virtual Memory Systems*, UMI Research Press, 1982.

[Mac87] Bruce J. MacLennan, in *Principles of Programming Languages*, Holt, Rinehart and Winston, 1987. 2nd ed.

[Mat80] R. E. Matick, Memory and Storage, in *Introduction to Computer Architecture*, Harold S. Stone (ed.), Science Research Associates, 1980, 205-274. 2nd ed..

[Mat86] R. E. Matick, Impact of memory systems on computer architecture and systems organization, *IBM Systems Journal 25*, 3/4 (1986), 274-305.

[McC69] A. C. McKellar and E. G. Coffman,Jr., Organizing Matrices and Matrix Operations for Paged Memory Systems, *IBM Systems Journal 12*, 3 (Mar. 1969), 216-237.

[Min70] Marvin L. Minsky, Form and Content in Computer Science , *J. ACM 17*, 2 (1970), 197-215. ACM Turing Lecture.

[Mow76] Abbe Mowshowitz, *The Conquest of Will: Information Processing in Human Affairs*, Addison-Wesley Publishing Company, 1976.

[Mye78] Glenford J. Myers, *Advances in Computer Architecture*, John Wiley & Sons, 1978.

[Pra75] Terrence W. Pratt, *Programming Languages: Design and Implementation*, Prentice-Hall, 1975.

[Rog75] J. G. Rogers, Structured programming in virtual storage systems, *IBM Systems Journal 14*, 4 (1975), 385-404.

[Say69] D. Sayre, Is automatic folding of programs enough to displace manual?, *Comm. ACM 12*, 12 (Dec. 1969), 656-660.

[Sed78] Robert Sedgewick, Implementing Quicksort Programs, *Comm. ACM 21*, 10 (Oct. 1978), 847-857.

[Sim69] Herbert A. Simon, *The Sciences of the Artificial*, The M.I.T. Press, 1969.

[Smi82] Alan Jay Smith, Cache Memories, *Computing Surveys 14*, 3 (SEPT 1982), 473-530.

[Sto78] Harold S. Stone, Sorting on STAR, *IEEE T-SE SE-4*, 2 (Mar. 1978), 138-.

[Sut75] John W. Sutherland, *Systems: Analysis, Administration and Architecture*, Van Nostrand Reinhold Company, 1975.

[Svo76] L. Svobodova, *Computer Performance Measurement and Evaluation Methods: Analysis and Applications*, Elsevier Computer Science Library, 1976.

[Tan76] Andrew S. Tanenbaum, *Structured Computer Organization*, Prentice-Hall, 1976.

[Tar87] Robert E. Tarjan, Algorithm Design, *Comm. ACM 30*, 3 (Mar. 1987), 205-212. Turing Award Lecture.

[Ver86] A. Inkeri Verkamo, Comparison of External Sorting and Internal Sorting in Virtual Memory, *Performance Evaluation 6*, (JUNE 1986), 135-145.

[WeA70] Alvin S. Weinstein and Stanley W. Angrist, *An introduction to The Art of Engineering*, Allyn and Bacon, Inc., 1970.

[Zad74] L. A. Zadeh, A new approach to System Analysis, in *Man and Computer*, M. Marois (ed.), North-Holland, 1974, 55-92.